Master's thesis

Elias Brattli Sørensen

Using Static Analysis to Detect Vulnerabilities in OpenID Connect Clients

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2020

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Elias Brattli Sørensen

Using Static Analysis to Detect Vulnerabilities in OpenID Connect Clients

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Abstract

OpenID Connect has become a de facto standard for managing authentication and authorization in Web applications. It is however challenging for developers to understand the protocol and securely implement a client application. Even using an SDK that helps them along the way, developers are responsible for doing data validation in a precise manner. The correctness of this validation can be ensured using security analysis and vulnerability detection tools.

Previous solutions on security analysis and tools for vulnerability detection of OpenID are mostly based on complex, formal models and comprehensive penetration testing frameworks that cover the whole protocol. These often require much work to understand, develop and use.

The objective of this thesis is to introduce a more developer-oriented way to ensure fewer vulnerabilities in such client applications. This thesis proposes (1) a pragmatic model of the authorization code flow, as a straightforward checklist targeted specifically at the concerns of the developer, and (2) a demonstration that relatively simple static analysis techniques, based on this model, can be used to find vulnerabilities related to the needed security checks.

The effectiveness of the analysis techniques is demonstrated experimentally on six open-source clients, of which four were found to have vulnerabilities. 20 vulnerabilities regarding incomplete or missing token validation were detected. The analyzer for token validation had a precision of 61%, recall of 100% and a true negative rate of 90%. Its precision may be improved further with a few weeks of engineering effort. More reliable metrics of its performance can be found by doing a large-scale empirical study.

Sammendrag

OpenID Connect has blitt en bransjestandard for å håndtere autentisering og autorisering i Web-applikasjoner. Likevel er det vanskelig for utviklere å forstå protokollen og implementere en klient-applikasjon på en sikker måte. Selv om de bruker en SDK som hjelper dem med detaljene, er utviklerne ansvarlige for å presist håndtere datavalidering. Sikkerhetsanalyse og automatiske verktøy for å finne svakheter kan bli brukt til å sørge for at denne datavalideringen er gjort skikkelig.

Tidligere løsninger på sikkerhetsanlyse og automatiske detekteringsverkøy for OpenID er for det meste bygget på komplekse, formelle modeller og helhetlige rammeverk for "penetration testing", som dekker hele protokollen. Det er ofte krevende å forstå, utvikle og bruke disse løsningene.

Målet med denne masteroppgaven er å introdusere en mer utvikler-orientert måte for å begrense mengden sikkerhetshull i klient-applikasjoner. Denne oppgaven presenterer (1) en pragmatisk modell av protokollflyten, formet som en direkte sjekkliste som er rettet mot det som angår utvikleren, og (2) viser at enkle statiske kodeanalyser som er basert på denne modellen, kan brukes til å finne svakheter relatert til disse sikkerhetsjekkene.

Styrken til analyseteknikkene presenteres eksperimentelt på seks klienter med åpen kildekode. Fire av disse har svakheter. 20 svakheter knyttet til usikker validering av "ID-tokens" ble avdekket. Analysen for validering av ID-tokens fikk en *precision* på 61%, *recall* på 95% og en falsk-negativ-rate på 90%. Presisjonen kan økes ytterligere med noen ukers ingeniørarbeid. Mer pålitelige metrikker kan bli funnet i en stor-skala empirisk studie.

Preface

Acknowledgment

This research would not have been possible without Kantega and NTNU, and the people helping me. Kantega has been an arena where I have been welcome to come and work, and the consultants there have been helpful and willingly participating in the study I did in my work preceding this thesis. I am thankful that I have had the opportunity to study computer science at NTNU, taking many interesting and educational courses that gave me the needed theoretical foundation.

Thanks the people who have given me good advice and proof-reading along the way. Thanks to Bjarte Østvold for your advice, our meetings have been really helpful.

I would like to give special thanks to co-supervisor, sparring partner and colleague from Kantega, Edvard Karlsen, and my supervisor Jingyue Li for their invaluable contributions to my work in encouraging and engaging discussions throughout the project.

Finally I want to thank my family and closest friends for your support and encouragement. I was able to complete this thesis thanks to you all!

Trondheim, June 19, 2020

Elias Brattli Sørensen

Contents

List of Figures

List of Tables

Glossary

Access token A credential understood by Identity Providers, that is used to grant access to clients.

Authorization The management of access to protected resources.

Authentication The identification of a subject like an end-user.

Client An application that connects with an Identity Provider for identity management.

Detector An class in FindBugs, which is an analyzer scanning Java classes for detecting a certain kind of bugs. FindBugs is formed of several detectors.

Identity Provider The server in OpenID Connect that manages the authentication of the end-user.

ID token A data item in OpenID connect given in the JSON Web Token format, that contains information about the user's identity as well as integrity-ensuring data.

JSON Web Token An open, industry standard method for representing claims securely between two parties like the Client and the Identity Provider.

OAuth 2.0 A distributed protocol for authorization.

OpenID Connect An identity layer on top of the OAuth 2.0 authorization protocol.

Penetration testing Testing a system for vulnerabilities by launching attacks and analyzing the outcomes.

Precision The fraction of warnings from an analyzer that were true vulnerabilities.

Recall The fraction of true vulnerabilities that were detected by an analyzer.

Relying Party Another term for Client in OpenID Connect.

Static analysis Automatic inspection, reasoning about the program's code without running it.

True Negative Rate The fraction of non-vulnerable code that was classified as non-vulnerable by an analyzer.

Acronyms

API	Application Programming Interface
IDE	Integrated Development Environment
IdP	Identity Provider
JWT	JSON Web Token
OIDC	OpenID Connect
RP	Relying Party
TNR	True Negative Rate
SDK	Software Development Kit

Chapter 1

Introduction

1.1 Motivation

OpenID Connect (OIDC) is becoming increasingly common in modern Web applications as a de facto standard for authentication and authorization with Single sign-on federation services. Developers may use well-known Software Development Kits (SDKs) for building a Relying Party (RP) in OIDC to connect their app with an Identity Provider (IdP). Examples of such SDKs are the Nimbus OAuth SDK [18] and Google OAuth Client Library [38, 39].

Even if these SDKs help the developer by encapsulating several difficult implementation details, the developer and the SDK still share a common responsibility in securing the RP application. The SDKs give tools for managing Web-specific features, and can provide and parse strong data types for the data delivered between the RP and the IdP. Still, the developer is responsible for establishing a trust relationship with the IdP, and correctly managing secrets and data that are needed to ensure integrity, confidentiality, and non-repudiation in the communication.

Listing 1.1 shows a code sample from an open-source Android-app project [102], where the developer has written code [1] verifying the ID Token using the Google library.

```java
boolean isValidIdToken(String clientId, String tokenString) {
    if (clientId == null || tokenString == null) {
        return false;
    }
    List<String> audiences = Collections.singletonList(clientId);
    IdTokenVerifier verifier = new IdTokenVerifier
                                    .Builder()
                                    .setAudience(audiences).build();
    IdToken idToken = IdToken.parse(new GsonFactory(), tokenString);
    return verifier.verify(idToken);
}
```

Listing 1.1: Incomplete ID Token verification in an open-source android app project [102].

[1]OIDCUtils.java in the Zop-App project: https://github.com/zopspace/zop-app/blob/fc7f9a9b6f9e0f18b89612ced49d67001aa61deb/app/src/main/java/fi/aalto/legroup/zop/authentication/OIDCUtils.java

While the example may look fine at first sight, there are several risks associated with this code. It lacks the following checks to satisfy the security requirements of the protocol specification [60, 62]:

- *Cryptographic signature* validation, which is important to ensure the integrity of the token.
- Verification of the *nonce parameter*, preventing replay attacks.
- *Issuer* validation, checking the identity of the IdP that issued the token.

The *audience* parameter is validated here, and the IdTokenVerifier on line 6 hides a default *Freshness* validation (which ensures that old or expired tokens are not used). This app is therefore vulnerable of being exposed to known threats like man-in-the-middle or replay attacks.

To understand which threats exist in the protocol and implementations of it, existing research has been doing security analyses. Security analyses of OpenID Connect can divided along to axes: one looks at formal security analysis and modeling of the protocol, or formal security testing [2, 33, 34, 49, 62, 80, 86, 89], while the looks at implementations of the protocol with automated vulnerability analysis or testing tools [11, 51, 54, 76, 95, 96, 97, 99]. These solutions generally seek to be comprehensive and tend to look at the threat models from the perspective of a hacker (or an attacker). Several of the automated vulnerability analysis tools require extensive work and configuration to use for discovering vulnerabilities in an application implementing the protocol, and detect vulnerabilities late in the software development life cycle.

Vulnerabilities in OpenID Connect can be considered a subset of *Access Control Vulnerabilities*, the fifth highest ranking risk according to the OWASP top 10 list of Web Application Security Risks [92]. There have been several known cases of data breaches due to insecure Single Sign-On implementations in the later years, like the Facebook breach in 2018 [55], where millions of access tokens were hijacked. Due to insecurely implemented Relying Parties lacking proper session management, adversaries could gain access to hundreds of websites outside of Facebook itself. Even though existing solutions have been made to automatically detect vulnerabilities, more must be done earlier in the development stage since several clients on the Web still have vulnerabilities in their production code.

1.2 Objectives

I hypothesize that easy-to-use incomplete static analyses can be used for mitigating vulnerabilities in Relying Party applications, early in the development stage. Simple-to-use static analysis tools that do not require any configuration are something developers like [88]. These simple analyses may mitigate vulnerabilities in a large portion of the more common security-critical steps in OpenID Connect, as the steps share similar (uncomplicated) structural and syntactic properties. The code structure for such critical steps is likely (or at least encouraged) to be relatively linear and simple [3], and vulnerabilities may consequently be quite easy to find.

The main objectives of this thesis are to:

- Provide a more pragmatic and developer-oriented way to look at implementation of OpenID Connect with OpenID connect SDKs,
- summarize the vulnerabilities that developers can introduce in implementing the *Relying Party* with such SDKs, and
- explore whether these vulnerabilities can be effectively detected with simple static analyses.

1.3 Research Questions

From the objectives, research questions were formed to direct the thesis. The research questions that this thesis seeks to answer are:

RQ1 What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?

RQ2 How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?

1.4 Contributions

This thesis proposes the following contributions related to RQ1:

- A pragmatic qualitative model of OpenID Connect, highlighting the RP developer's concerns (Chapter 5.1).
- The *development checklist*, which is a step-by-step development recipe rooted in a thorough analysis of the protocol and the OpenID Connect SDKs (Chapter 5.2).
- An overview of the implementation errors that cause potential vulnerabilities by breaking the rules of the model (Chapter 5.3).

The suggested contributions with regard to RQ2 are as follows:

- Three simple, "peephole"-based static analyses explained in Chapter 6:
  - The simplest *Immediate Code Smell Detection*, which detects a single JVM bytecode instruction that indicates anti-patterns.
  - The moderately simple *Co-existing Invocation Enforcement*, which uses the co-occurrence of instruction patterns to infer absence of needed secure code checks.
  - The slightly more complex *Static Control Flow Check*, which uses simple patterns in the basic blocks of the program's control flow graph, to detect improper responses to certain security steps.
- Code examples of the vulnerabilities and their patterns related to each of the implemented static analyses, with vulnerability-specific detection strategies (Chapter 6.2).

- An overview of which OIDC vulnerabilities can be covered by which of the different analyses (Chapter 6.3).
- The first static analysis detecting ID token verification vulnerabilities, the *Improper ID token verification detector* [2] (See Chapter 6.4.3).
- An experimental validation on six open-source Java web applications using OpenID Connect, demonstrating the effectiveness of the analyses (Chapter 7).
  - Four of the six applications had vulnerabilities. A total of 20 vulnerabilities were found, with 19 false positives, 147 true negatives and one false negative.
  - The recall of the tool in total was 95%, which means most of the known vulnerable code was discovered. The Improper ID token verification detector had a recall of 100%.
  - The tool also had a precision of 51% in total, and 61% for the *Improper ID token verification detector.*
  - The true negative rate was 89%, meaning that 9 out of 10 non-vulnerable cases were correctly predicted as negatives.

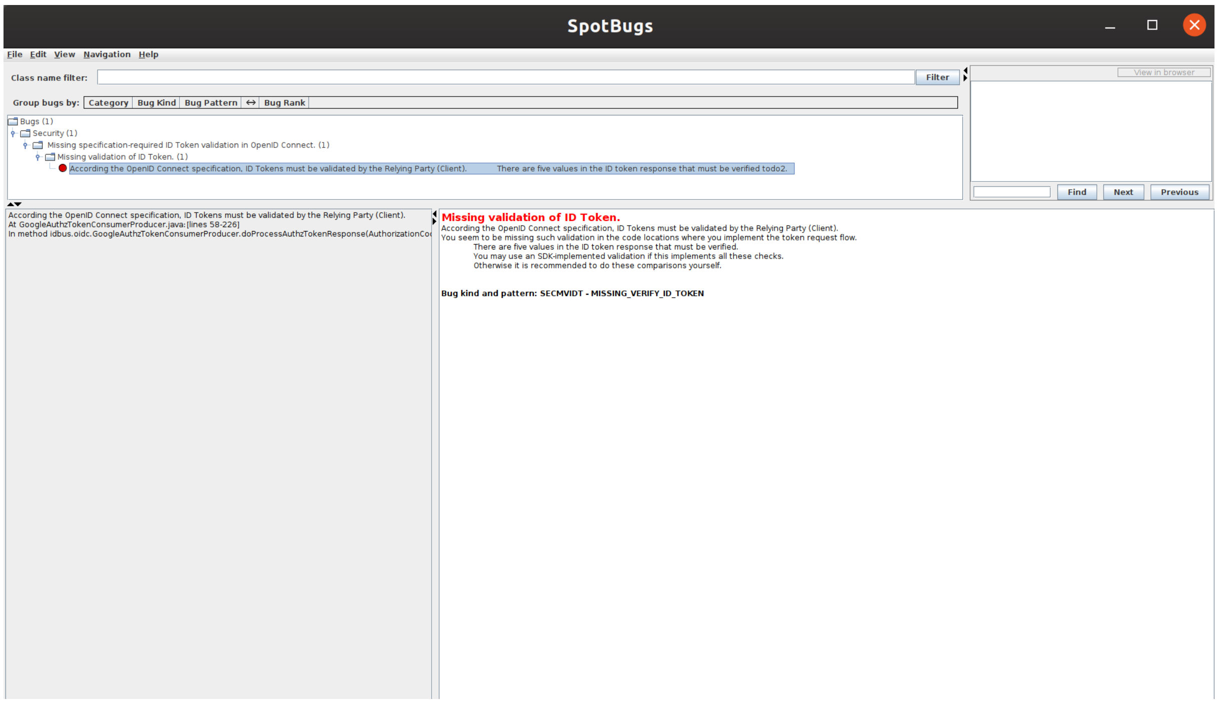
This thesis explores the possibility that relatively simple and explicit static analysis techniques can be used to find vulnerabilities in OpenID Connect, such as the ones in Listing 1.1.

The simple process of the analyses is demonstrated with an example. For detecting the incomplete verification in Listing 1.1, the analyses could use something like the following process:

1. This is a token verification method in OIDC. Another method with token request called this method, and the method name and signature indicate token verification.
2. Here we expect that at least these *n* verification steps in the checklist are performed.
3. If any one of these steps is absent in the code, raise a warning.
4. The warning informs the developer of the risks associated with not performing these checks.

The analyses are added to the Find Security Bugs plugin, which is a popular easy-to-use static analysis tool for detecting security bugs in Java. The tool comes as an IDE plugin, which makes the analyses easily accessible to developers implementing the protocol in real-life web applications. It may also be used in the graphical user interface provided by SpotBugs, giving results like shown in Figure 1.1, where a vulnerability of missing ID token validation is raised for a method.

[2]This claim is supported in comparison to related work in Chapter 8.2.2


SpotBugs
File Edit View Navigation Help
Class name filter:
Filter
Group bugs by: Category | Bug Kind | Bug Pattern | ↔ | Bug Rank
View in browser
Find
Next
Previous

```
Bugs (1)
  Security (1)
    Missing specification-required ID Token validation in OpenID Connect. (1)
      Missing validation of ID Token. (1)
        According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).        There are five values in the ID token response that must be verified todo2.
```

```
According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).
At GoogleAuthzTokenConsumerProducer.java:[lines 58-226]
In method idbus.oidc.GoogleAuthzTokenConsumerProducer.doProcessAuthzTokenResponse(AuthorizationCo
```

Missing validation of ID Token.

```
According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).
You seem to be missing such validation in the code locations where you implement the token request flow.
        There are five values in the ID token response that must be verified.
        You may use an SDK-implemented validation if this implements all these checks.
        Otherwise it is recommended to do these comparisons yourself.


Bug kind and pattern: SECMVIDT - MISSING_VERIFY_ID_TOKEN
```

Figure 1.1: SpotBugs GUI showing a vulnerability: The file *GoogleAuthzTokenConsumer* is missing validation of the ID token.

Vulnerabilities are detected by tailoring the checks based on the protocol flow, and *checklist* over what steps are needed to ensure a secure RP.

This way the developer is instructed directly of the risks associated with the code flaws in their security checks, while they still are in the context where the check is relevant. The focus here is vulnerabilities in code calling OpenID Connect SDKs, meaning vulnerabilities that developers introduce when they write code that interfaces with these SDKs. To be clear, the analyses are not concerned with looking for vulnerabilities in the SDKs themselves.

1.5 Structure of the Thesis

The rest of the thesis is structured as follows: Chapter 2 goes through background for authorization (Access Control), the security protocols OAuth 2.0 and OpenID Connect, theory about program analysis techniques and tools, and an overview of the architecture of the static analysis tool Find Security Bugs. Chapter 4 shows the approach and overall research strategy. Chapter 5 contains the analysis results and qualitative model that emerged from RQ1, while Chapter 6 explains the implementation answering RQ2. Chapter 7 shows an experimental evaluation of the implementation. The results are discussed in Chapter 8. Finally, Chapter 9 contains the conclusion and recommendations for further work.

Chapter 2

Background

This chapter goes into the theoretical background with explanation and definitions of topics that are used in this thesis. OpenID Connect and OAuth 2.0 are used to ensure authentication and authorization. Therefore Section 2.1 goes through Access Control (Authorization), with an overview of common vulnerabilities. Section 2.3 shows how the authorization protocol OAuth 2.0 works, and Section 2.4 explains the workings of the authentication protocol OpenID Connect. Then comes an insight into program analysis techniques in Section 2.5, and the way program analysis tools are evaluated in Section 2.6. The abilities and architecture of the static analysis tool Find Security Bugs is explained in Section 2.7.

Sections 2.1, 2.2, and 2.5 contain theoretical background that was mainly outlined during the specialization project preceding this thesis [87].

2.1 Access Control

Access Control (Authorization) within information security is according to Benantar [9, p. 1] concerned with a system's ability to limit computing resources to be exclusively accessible to authorized entities. Typical terms used in this topic are *user*, *principal*, *subject* and *object*. A *user* is typically defined as a person, thus an external entity, interacting with a system. The usual user account contains information about both authentication and authorization. Meanwhile, a *principal* is a reference to the internal representation of an entity in a system. While a user may be associated with several principals, a principal refers to one unique user. *Subjects* identify the running processes in a system, i.e. the programs in execution. A principal may be associated with multiple subjects, as they can have several processes running concurrently. An object in a system is often also referred to as a *resource*. A resource may be a computing service of some kind, but could also be a passive information entity like a file or a database record [9, p. 9].

2.1.1 Access Control Models

A *security policy* is the set of rules in an organization that define who has access to which resources. Thus, the security policy describes the *protection states* within a system. Definitions of access control policies are collected in a set of paradigms or models [9, pp. 25–26]. The main access control models are Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC). Another, newer model is Attribute-Based Access Control (ABAC) [66]. These models have the following characteristics:

Mandatory Access Control, which is based on information sensitivity within resources, with a formal authorization. Subjects are restrained from setting security attributes on a resource, and cannot pass on their access, hence the model is mandatory.

Discretionary Access Control, which is based on the identity of subjects, and what information they need to know, in addition to group affiliation. A subject with a set of access permissions may pass their access on to other subjects, hence the model is discretionary.

Role-Based Access Control, which is based on roles within an organization, that are projected on to users and groups. Roles include collections of subjects within the organization that have a common need for access in order to perform their tasks. Access levels or a set of permissions is formally defined for a role or group, and member subjects inherit permissions.

Attribute-Based Access Control, which is based on properties of an information exchange. The exchange may include the resource requested, the identified attributes of the requesting entity, or the context of the requested action or the exchange. Attributes in the context may be time of day, location and currently evaluated threat level.

2.2 Access Control Vulnerabilities

Access control vulnerabilities have long been among the highest ranked risks in Web applications. The vulnerabilities are common, because they are not easily targeted with automatic tests and because functional testers do not necessarily have the skills to properly test access control mechanisms. Detection of access control vulnerabilities is not considered an easy task for automated vulnerability detection tools. In an application the system enforces the policy in such a way that users are restricted from acting outside of their intended permissions [92].

2.2.1 Clarification of terms and definitions

Due to rapidly updated vulnerability classifications by both OWASP Top 10 [92] and Common Weakness Enumeration (CWE) [91], it is hard to get a sound classification of the scope in dated literature. The classifications that were made back then, no longer have a clear definition in the newest updated lists. Inconsistent usage of terms makes it challenging to properly classify the vulnerabilities that fall within the scope of an analyzer. Different researchers tend to use their own definitions and understandings of the same terms, or use the existing references to classifications. Access control is here viewed as any mechanism explicitly or implicitly involved in controlling access to data in a given system. Here, the following definitions are proposed to reason about the term “access control vulnerability”:

Definition 2.2.1. Data Leakage (DL) If some observer O can learn a piece of information I from a software system S, and O is not supposed to be able to learn I, S has a Data Leakage.

Definition 2.2.2. Explicit Access Control Vulnerability (E_{ACV}) Explicit access control vulnerabilities are cases where the program source code explicitly fails in enforcing concrete, program-specific access control rules, causing a data leakage.

Definition 2.2.3. Implicit Access Control Vulnerability (I_{ACV}) Implicit access control vulnerabilities are any, potentially highly subtle, property of the program or software system that can cause data leakage.

Definition 2.2.4. Access Control Vulnerability ($ACV = I_{ACV} \cup E_{ACV}$) Access control vulnerabilities are the union of all explicit and implicit access control vulnerabilities, causing a subset of all possible data leakages.

Here the relation is that $DL \supseteq E_{ACV} \cup I_{ACV}$, meaning that an DL may entail subtle weaknesses that fall far beyond the scope of typical web-based access checks. Therefore the further definition for ACV needs some fine-tuning, and for the purpose of this project, mentions of ACV will be therefore limited to $ACV \equiv E_{ACV} \cup I_{ACV}$

2.2.2 Examples

Examples of access control vulnerabilities common access control vulnerabilities in Web applications include:

- Metadata manipulation, where attacks are done through tampering with or replaying access token or cookie, or manipulation of a hidden field to elevate privileges.
- Privilege escalation, for example when someone acts as a user without being logged in, or manages to perform admin level actions as a normal user.
- Modification of application state to bypass access control checks. This includes modification of internal app state, URL, the HTML page or a custom API attacking tool.

- Improperly restricted database access, allowing someone's primary key to be changed into another subject's record. This may allow other subjects to view or edit another account.
- Cross-Origin Resource Sharing (CORS) misconfiguration which allows unauthorized access to API endpoints. This opens for Cross-site request forgery (CSRF) attacks, which may lead to privilege escalation. In CSRF attacks, a malicious server forges a request pretending to be an honest party.
- Improper enforcement of POST, PUT and DELETE requests.

Other vulnerability classes in OWASP Top 10 as well as the CWE lists are used to define different vulnerability classes can be considered relevant to access control, as for example cross-site scripting (XSS) may lead to privilege escalation.

Modern Singe Sign-On (SSO) protocols seek to solve several of the issues with more classical Web-based vulnerabilities by leaving access management to a designated server, thus separating identity management and delivery of resources. Even if several traditionally rooted attacks become irrelevant with modern SSO, attacks with similar characteristics can still be applied to applications implementing such protocols. Data must still be sanitized, and the communications are still based on web requests.

Additionally, traditional protection mechanisms that work with a client-server model may be circumvented if the application uses single sign-on, thus introducing novel attack surfaces. Still, consequently if authorization is broken due to a flaw in the implemented authorization protocol, it can be considered as an access control vulnerability as data was leaked.

To better understand which vulnerabilities will apply when using single sign-on, Sections 2.3 and 2.4 go through the specification of the authorization protocol OAuth 2.0 and its extension for identity management, OpenID Connect.

2.3 OAuth 2.0

A modern and widely used authorization protocol is known as OAuth 2.0, which is proposed in RFC6749 [41], replacing and obsoleting the original OAuth protocol. The protocol was introduced as a means to address the issues with traditional client-server authentication models, which suffer from limitations like the inherent weaknesses of password security, and problematic management of third-party resources and the access they have to the restricted resources on the server. The issues are addressed by OAuth through the introduction of an authorization layer, and the separation of roles. In OAuth the four roles are defined:

- **Resource Owner** or end-user if it is a person, which is an entity that grants access to a protected resource.
- The **Resource Server**, that hosts the protected resources. The server accepts and responds to protected resource requests by issuing access tokens.

- The **Client**, an application that uses the resource owner's authorization and makes protected resource request on their behalf.
- The **Authorization Server**, responsible for issuing access tokens to the client after the resource owner has been successfully authenticated and obtained authorization.

Meanwhile there are several data items that form the security properties of the protocol. The protocol relies on the following important credentials used in the requests:

- **Authorization Grant:** A credential representing that the resource owner has given consent allowing the client to obtain an access token.
- **Access Token:** A credential used to access protected resources, which is a string representing an authorization issued to the client. This token is an abstraction layer that replaces different authorization constructs with a single token understood by the resource server.
- **Refresh Token:** A credential used to obtain new access tokens when they expire or are invalidated. This is an optional item to include together with the access token when first prompted for tokens.

There are four authorization grant types that are defined in the protocol:

- **Authorization code grant** An authorization code is obtained with the authorization server used as an intermediate. The resource owner is redirected to the authorization server, which authenticates the resource owner and redirects them back with a code. A security advantage here is that the resource owner's credentials never are shared with the client, and that the code never is exposed through the user agent.

- **Implicit grant** The implicit grant is a simplified version of the flow used in the authorization code grant. Instead of a flow with round trips, the client gets an access token directly from the authorization server, effectively skipping the step that gives a code grant. This grant is optimized for clients that run directly in the browser (therefore using a language like Javascript). This flow introduces some security risks that must be considered against efficiency.

- **Resource Owner password grant** The password credentials of the resource owner are used directly as an authorization grant to obtain an access token, skipping the round-trip where an authorization code is issued. The client does not need to store the resource owner credentials, as these are used only once and can be replaced with a long-lived access or refresh token.

- **Client credentials grant** The client credentials are used directly as an authorization grant, effectively removing the resource owner from the picture. This grant can be used when the authorization scope is limited to protected resources that belong to the client.

One of the advantages of this protocol is in the way the resource server only has to understand and validate access tokens when issuing protected resources to various subjects, instead of having to handle various other authorization constructs. Otherwise a server would have to understand an authorization construct, where the access is defined by the resource owner directly authenticating with her username and password. OAuth is an abstraction layer that allows for more flexible authorization rules, where the token can get a specific duration of access and possibly a more restricted access than the authorization grant that was used to obtain the token.

2.3.1 Authorization flow

The authorization code grant flow in the protocol is illustrated in Figure 2.1. There are four different grant types defined: authorization code, implicit grant, client credentials and resource owner password credentials. Requests from the authorization server to the client are redirection-based.

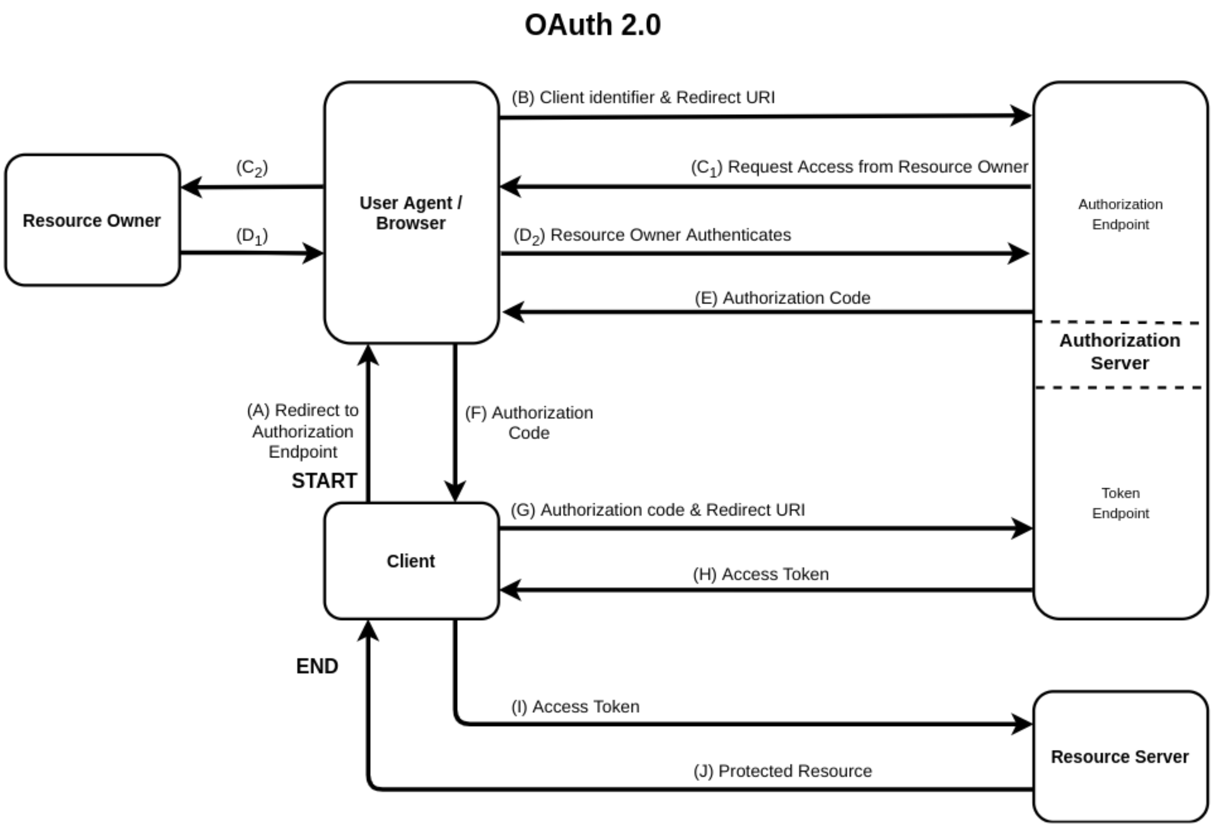


Figure 2.1: The authorization code flow in OAuth 2.0 has 10 steps from the client asking for access via the authorization server, so that the protected resource can be obtained with an access code. Steps C and D are broken into two parts, illustrating the interaction between the browser and the resource owner.

The following steps are included in the illustrated authorization flow in Figure 2.1, with the client seeking to access a protected resource at the resource server:

(A) The flow initiated by the client, redirecting the resource owner's browser to the authorization endpoint with a set of parameters.
(B) The included parameters from the client through the browser are *client identifier*, *requested scope*, *local state* and the *redirection URI*, which is the location the user agent is redirected after access is granted. The browser presents the authorization endpoint to the resource owner.
(C) Access is requested of the resource owner via the browser.
(D) The authorization server authenticates the resource owner, who either grants or denies access.
(E) With granted access, the redirection URI from step B is used to send the browser back to the client. Authorization code and the local state provided in the URI in step B are included as parameters in the redirection URI.
(F) Client receives the authorization code as the browser is directed back.
(G) The client requests to get an access token by contacting the token endpoint providing the authorization code, hence it authenticates with the authorization server. The redirection URI returned in step C is included for verification.
(H) The authorization server validates the authorization code, authenticates the client and verifies that the parameters in the redirection URI matches the URI used to redirect the client during step C. It returns an access token, and may optionally return a refresh token.
(I) Having obtained an access token from the authorization server, the client can finally request the protected resource from the resource server.
(J) The resource server validates the access token, providing the protected resource if the token is valid.

Note that the last two steps, I and J, are optional parts of the flow, and not encapsulated by the standard.

Authentication request parameters

In authorization requests, the client adds a specified set of parameters to the query component of the URI:

- *response_type*: Denotes what to expect in the response. This must be set as "code".
- *client_id*: The unique identifier of the client, which is known by the authorization server.
- *redirect_uri*: An encoded URI with the location to which the resource owner will be redirected after authenticating with the authorization server.
- *scope:* The scope of the access. It represents a limitation to what kind of data the access token can be used to obtain.
- *state:* An opaque string that is used to maintain a session state between the request and the callback response. This value protects against Cross-site request forgery (CSRF) attacks.

The basic data transfers in the authorization code flow can then be illustrated by looking at example HTTP requests. An authorization code request built by the client may look like shown in Listing 2.1 (Step A). The client redirects the resource owner to the location in the URL (Step B):

```
GET https://authorizationserver.domain.com/authorize
?response_type=code
&client_id=abc
&redirect_uri=https://org.client.com/callback
&state=xyz
```

Listing 2.1: Step A-B: URL format for an authorization code grant request.

The authorization server then responds with a callback request after performing step C, authenticating with the resource owner. It then redirects the resource owner back to the redirect_uri. A callback response may be structured like in Listing 2.2

```
HTTP/1.1 302 Found
Location: https://org.client.com/callback?
code=Splxl0BeZQQYbYS6WxSbIA
&state=xyz
```

Listing 2.2: URL format for an access token response.

Then the client validates the *state* parameter in step F. Upon success, proceeds to step G, and builds a token request using the authorization code it received. A token request typically looks like shown in Listing 2.3. This time instead of redirecting the user, the client directly contacts the authroization server on a back channel, leaving the browser out of the picture.

```
POST https://authorizationserver.domain.com/token?
    grant_type=authorization_code
    &code=Splxl0BeZQQYbYS6WxSbIA
    &client_id=abcde
    &client_secret=Xpbxlklk12WRlkoP
    &scope=api.read api.write
    &redirect_uri=https://org.client.com/callback
```

Listing 2.3: URL format for an access token response.

After this, in step H, the authorization server gives a token response after receiving the code. Token responses responses may come in the format shown in Listing 2.4:

```
HTTP/1.1 200 OK
 Content-Type: application/json
 {
   "access_token":"2YotnFZFEjr1zCsicMWpAA",
   "token_type":"bearer",
   "expires_in":3600,
   "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
   "scope": "api.read api.write"
 }
```

Listing 2.4: An access token response in the JSON format.

The authorization flow is designed to ensure the access control of an application. As such, vulnerabilities introduced in the implementation of the protocol would consequently be classified at access control-related vulnerabilities. While OAuth is designed to issue a client access to protected resources, its design does not properly work to handle the integrity of the data if used to do authentication where identity information must be obtained.

2.4 OpenID Connect

The authentication protocol OpenID Connect (OIDC) is a layer build on top of of OAuth 2.0 to handle identity management, taking the role as an industry-standard Single Sign-On (SSO) protocol [60]. One important factor in this identity layer is that other abstractions are used as optional terms to refer to what are somewhat the same roles. While the terms defined in the OAuth 2.0 standard are also used in OIDC, an additional set of terms are defined to manage the identity layer.

The OIDC roles are either new additions, correspond to, or are a subset of the OAuth roles. These roles are shown in Table 2.1.

Table 2.1: An overview of how OAuth roles are extended or referred to with different terms in OpenID Connect.

OAuth 2.0 Role	**OpenID Connect Role Addition**
Resource Server	The *Userinfo Endpoint* is exposed from the Resource Server as a protected resource giving information about the End-User in response to and Access Token.
Client	*Relying Party* refers to a Client application which requires Claims and End-User authentication from an OpenID Provider.
Authorization Server	*OpenId Provider* (OP), in the industry often referred to as *Identity Provider* (IdP), is an authorization server capable of authenticating the End-User and providing Claims about the Authentication event and the End-User to a Relying Party.

There are also more data artifacts and flows that are defined in the identify layer abstraction of the protocol. OpenID Connect introduces a set of request parameters in addition to the ones described in OAuth (See Section 2.3.1). Among these is the *nonce* value, which is a randomly generated string value. This value and the *state* value defined in OAuth serve similar purposes. The state value comes in the callback request with the authorization code in step F in the flow (See Figure 2.3), and must be verified by the relying party before the token request is initiated. It binds the authentication request to the callback authentication response. The nonce value comes with the token response, and ensures replay attack protection by binding the authentication request with the token response.

In addition, the following concepts are introduced in OpenID Connect:

- **ID Token:** A token that contains identifiers of the end-user as well as identifiers of the IdP and integrity timestamp. It also contains the nonce value, which binds the token response to the initial authentication request. This token is sent in the token response together with the access token. ID tokens come in the JSON Web Token (JWT) format, which is defined in RFC7519 [45].

- **Standard Scopes:** A standard set of scopes are defined to specify what identity information is available in a request. This information typically includes *profile* and *email*. OIDC requests must always include the *openid* scope value.
- **Claims:** Claims are specific sets of information about an entity, typically the identity information of a user.
- **Discovery:** The Discovery process is used to establish a trust relationship between the relying party and the Identity Provider. The relying party sends a request to the */.well-known/openid-configuration* endpoint at the IdP, and receives a JSON document which is called the Discovery document. This document forms a contract, and contains values that are used to ensure the integrity of the communication. The Discovery process is described in its own document which was publish alongside the OIDC specification [61].

A chart very similar to the model of OAuth (in Figure 2.1), is shown below in Figure 2.2.

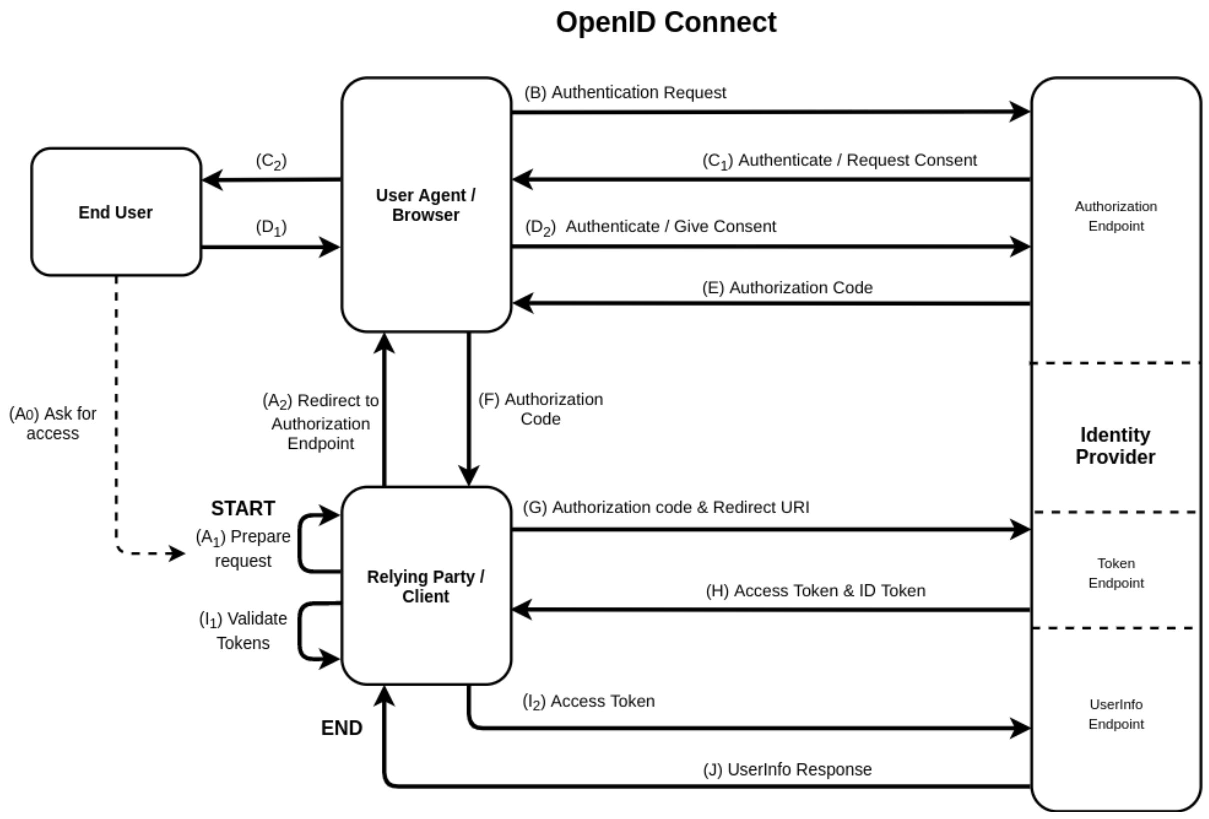


Figure 2.2: The authorization code flow in OpenID Connect is quite similar to the authorization flow in OAuth (Figure 2.1) as it is built on top of the authorization flow. The main difference lies in some different abstractions, otherwise we see the same 10 steps with added sub-steps. This model is based on the flow described by Navas and Beltrán [62].

Like Figure 2.2 shows, the main steps in the authorization flow are essentially the same for OpenID Connect as for OAuth 2.0 (Shown in Figure 2.1), with the round-trips redirecting the end-user. The details that differ mostly from OAuth have been expanded to sub-steps, which illustrates more of the critical validation events that are included in OIDC. Additionally, the responses contain IDs-specific data instead of the generalized terms that are present in OAuth.

Step A consequently consists of three sub-steps:

- (A_0) The end-user is the entity that naturally prompts the client to initiate the flow.
- (A_1) The client prepares the request. In this step it is critical to include the proper parameters, and mistakes here may compromise the client.
- (A_2) When the request is ready client redirects the user to to the authorization endpoint.

Steps H and I are also expanded to highlight the validation events that are important to ensure the integrity of the data sent between the entities.

- (G) The state parameter in the callback request is validated.
- (H) After verifying the authorization code and redirect URI, the token endpoint responds with an Access Token and ID token, and optionally a refresh token in addition.
- (I_1) The RP has to validate the access token and ID token it received from the token endpoint. Here the developer must remember to implement a specific flow, metadata-specified algorithms and check various data entries to ensure the validity of the tokens. This requires carefully implemented code, and is easily susceptible to errors as it relies on the developer.
- (I_2) When the tokens are validated, the RP can finally request the UserInfo endpoint for the protected user resource, passing the access token that is associated with the ID token.

Another perspective of this flow is presented in Figure 2.3, where the order of events flows downwards in the chart.

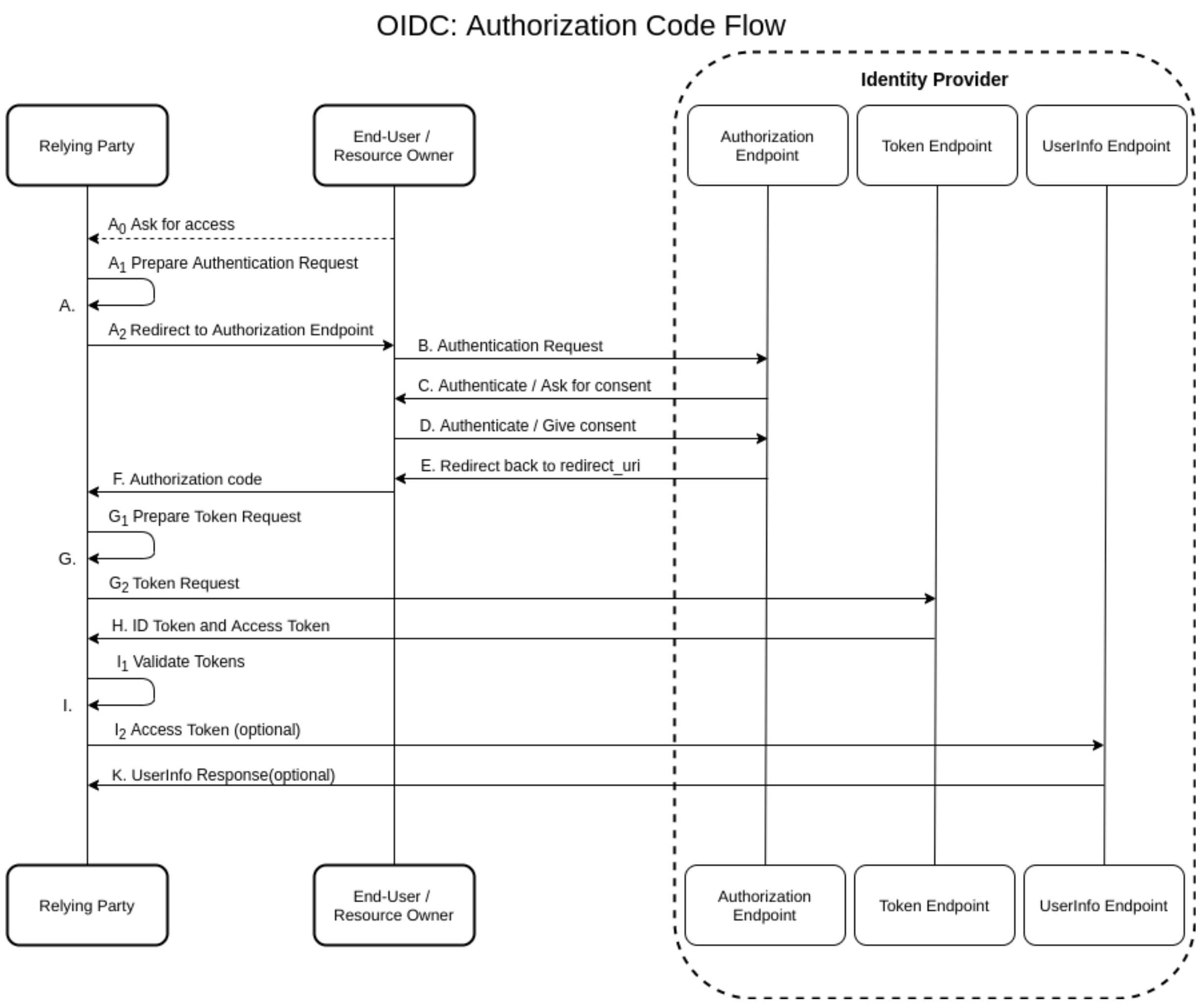

OIDC: Authorization Code Flow
Identity Provider
Relying Party
End-User / Resource Owner
Authorization Endpoint
Token Endpoint
UserInfo Endpoint
A₀ Ask for access
A₁ Prepare Authentication Request
A.
A₂ Redirect to Authorization Endpoint
B. Authentication Request
C. Authenticate / Ask for consent
D. Authenticate / Give consent
E. Redirect back to redirect_uri
F. Authorization code
G₁ Prepare Token Request
G.
G₂ Token Request
H. ID Token and Access Token
I₁ Validate Tokens
I.
I₂ Access Token (optional)
K. UserInfo Response(optional)

Figure 2.3: The authorization code flow in OpenID Connect in a sequence chart. This chart is based on the flow described by Navas and Beltrán [62].

Token validation

Validation of tokens is one of the critical features in the protocol. During validation there are several key steps that must be implemented correctly by the client developer. Table 2.2 shows the mandatory parameters in ID tokens. The ID token may contain a number of other claims that have more identity information.

Table 2.2: The most common ID Token parameters described [62].

Parameter	Description
iss	*Issuer Identifier* for the issuer of the token, the IdP in format of a case sensitive URL, using HTTPS.
sub	*Subject Identifier*, a unique and never reassigned identifier for the end user within the IdP.
aud	*Audience* the token is issued for. An array of case sensitive strings that at least must include the client_id of the RP that sent the Authentication Request.
exp	*Expiration* time for the ID token in Unix Epoch time. The ID token must not be validated after this time.
iat	*Issued At Time*, when the ID token was issued in Unix Epoch time. This limits amount of time nonces need to be stored.

2.4.1 Token validation

In the authorization code flow the ID Token must be validated in order to ensure integrity. A correctly implemented ID token validation in OIDC must include the following steps [60, 62]:

- **Token parsing**: Received tokens must to be parsed into data objects so that they can be processed further. All the required parameters must be present in the response. If any of them is missing an appropriate error message must be produced. Following the specification, any parameters that are not understood must be ignored.
- **Origin verification**: The RP receiving a token must validate the iss parameter, which is the unique identity of the IdP. It must also check that the corresponding shared secrets, keys, certificates, and other parameters are available and updated. These are needed to perform further cryptographic verification.
- **Audience verification**: A token is intended for a single RP. Hence the aud parameter should be checked for the correct value (the $client\_id$ value issued from the IdP during registration).
- **Freshness validation**: Validation of the token's age to detect expired tokens. Parameters such as *exp* and *iat* enables this validation. This is essential to avoid replay attacks.
- **Session validation**: The RP receiving a token must validate that the received nonce parameter matches the one that was issued initially.

- **Cryptographic validation**: This task involves the verification of signatures, and is usually the more time and resource consuming task. The cryptographic material (key, cipher, etc.) belonging to the legitimate IdP must be used.

In addition to this, timing attacks may leak potentially useful information to an attacker. If the code paths taken by successful or unsuccessful validation processes differ greatly, the attacker may learn much about how the validation is structured. It is suggested to terminate the processes and send an error message as soon as an error is found. All responses should take similar amounts of time, whether they are successful or not [62].

Access Tokens may also be validated, but is considered an optional step in the Authorization Code Flow. However using the Implicit Flow, the client *must* validate the access token [60] as the cryptographic integrity must be maintained as data has passed by untrusted actors over a less secure connection.

2.5 Program analysis

This section contains background information that was found during the project preceding this thesis [87].

One way to mitigate security vulnerabilities is through security testing. Security testing is the act of testing an application's behavior with regards to security. The testing can be done manually by a person, in a combination, or automatically in written software tests or program analysis. There are several kinds of security testing, and the most common and distinct ones are *black-box* and *white-box* analysis. With black-box analysis, the tester evaluates the security behavior without having access to the source code. This way, if is confirmed whether the information that is available to outside malicious entities is enough to perform malicious actions. White-box analysis happens with access to the source code, hence a deeper access to information of the application's behavior [75].

Program analysis tools (PATs) reason about a program's behavior with regards to properties such as security vulnerabilities. Program analysis can be done as static analysis (a kind of white-box testing), in which the program code is analyzed and the behavior is reasoned about without execution, dynamic analysis (black-box), where the code is executed and its run-time behavior is analyzed, or in a combination of both, which is called hybrid analysis [64].

Several common techniques used in program analysis imply a trade-off between the accuracy and the computational efficiency of the analysis [75]:

Flow-sensitive analysis (Section 2.5.1) reasons about the program with the control-flow graph (CFG). It is usually accurate, at the expense of also being time consuming.

Path-sensitive analysis considers path throughout the program that are valid. Variable values and booleans in conditionals or loops are reasoned about, so that execution branches that are not possible can be pruned. Like flow-sensitivity, path-sensitivity implies accuracy at a computational cost.

Context-sensitive analysis takes into account things like global variables and parameters of a function call, which form what can be considered the context. Context-sensitive analysis is also known as *inter-procedural* analysis, which comes as a contrast to *intra-procedural analysis*, that uses no context when analyzing a function. Context-sensitivity implies a larger computational cost, but with a significant gain in accuracy compared to intra-procecural analysis.

Pattern matching analysis uses simple linear code scans in a file to power a state machine, looking for certain patterns of instructions in the code (like the invocation of a certain type). Heuristics can be used to approximate control-flow of the program. It is very fast and requires little memory, at the expense of accuracy [36, 44].

2.5.1 Control Flow analysis

In control flow analysis the purpose is to extract information about which of the basic blocks have paths to other basic blocks [64, p. 10]. Control flow analysis is often also expressed as *constraint based analysis*. The flows are represented in a control flow graph (CFG), which is a directed graph where nodes represent the basic blocks and edges represent paths in the control flow. A basic block is set as a linear set of instructions which have only one entry point and one exit point, which represent the start and end of the graph [4, 5]. Two examples of control flow statements are if-else statements and while-statements, which are illustrated in Figure 2.4 below. Another common data structure is a call graph, which differs from a CFG in the way that it does not include returns.

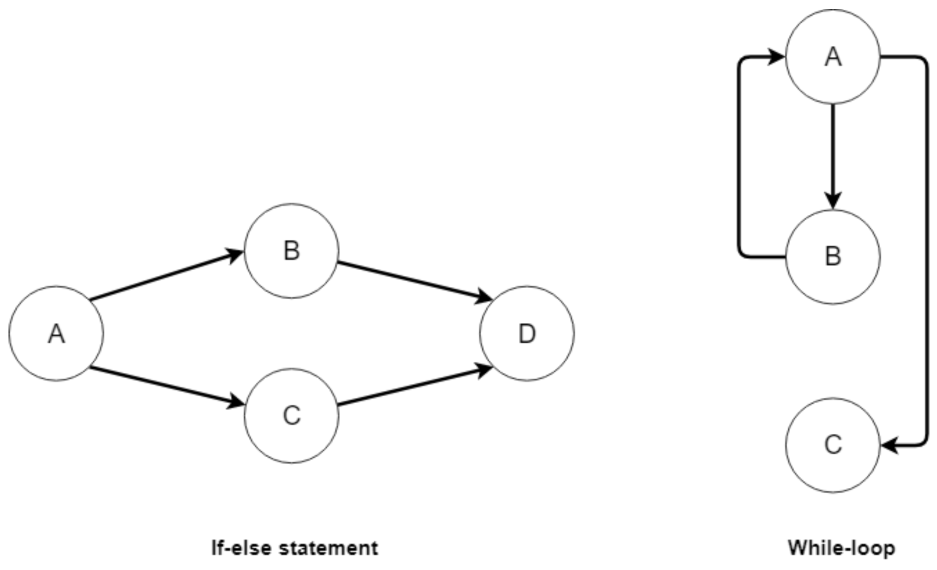


Figure 2.4: CFG examples of if statement and while loop.

2.5.2 Data flow analysis

In data flow analysis, it is common to regard the program as a graph. The program components in a certain abstraction can be seen as "elementary" or basic blocks. These blocks form the nodes in a page graph. The edges describe the way control may pass between basic blocks [64, p. 5].

Figure 2.6 shows how control flow and data flow may pass through an arbitrary CFG. Data may flow independent of the control due to global data structures [42]. Both direct data flows E. G. $(C \rightarrow D)$, $(D \rightarrow E)$ and indirect data flows like $(B \rightarrow A)$, $(E \rightarrow D)$.

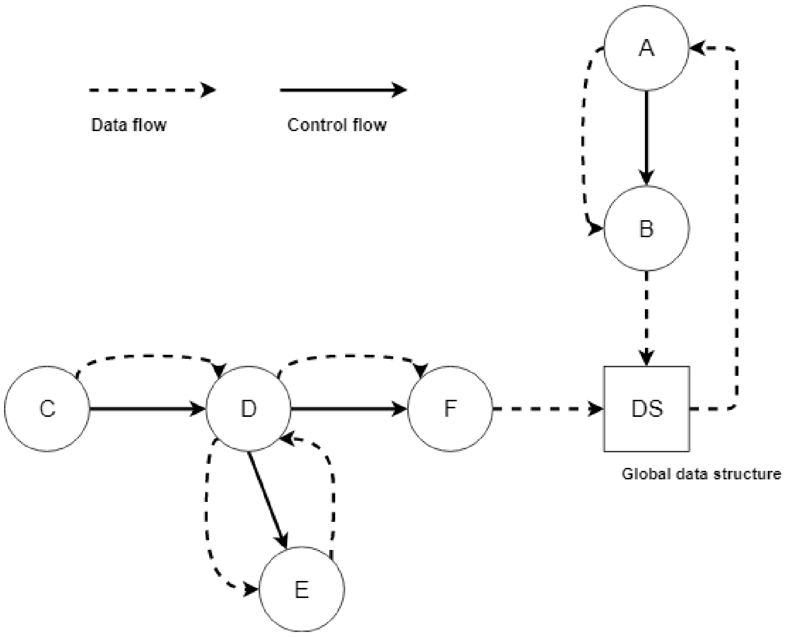
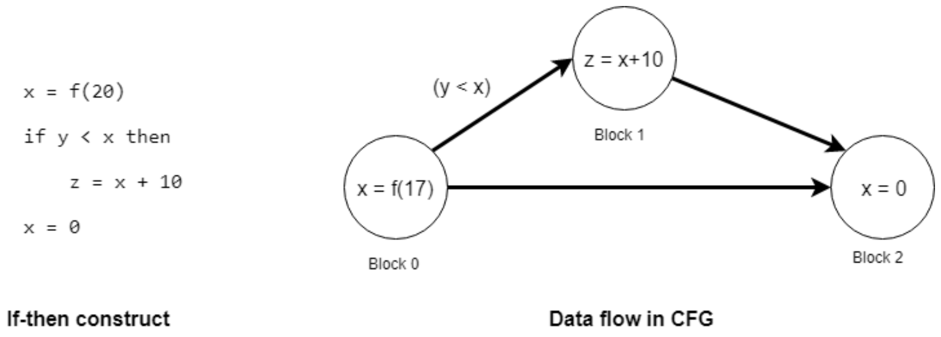


Figure 2.5: CFG illustration of data flow and control flow

Figure 2.6 illustrates a more concrete example of the data flow in a CFG of a simple if-then statement [21, p. 488]:



```
x = f(20)
if y < x then
    z = x + 10
x = 0
```

Figure 2.6: Simple if-then statement with corresponding data flow in CFG

2.5.3 Language processing

Finite state automata

Finite state automata (FSA), or state machines, are seen as a flexible tools. FSA can either be viewed as something that defines a language (i.e. a regular language), or defining a class of graphs. The construction of an FSA contains a finite set which is the alphabet Σ, a finite set of states Q, the initial state $i \in Q$, the set of final states $F \subseteq Q$, and the set of edges E. This forms a 5-tuple $\langle \Sigma, Q, i, F, E \rangle$. Figure 2.7 below shows two examples of simple finite state automata representing all the multiples of two and three as binary numbers [78, pp. 3–5].

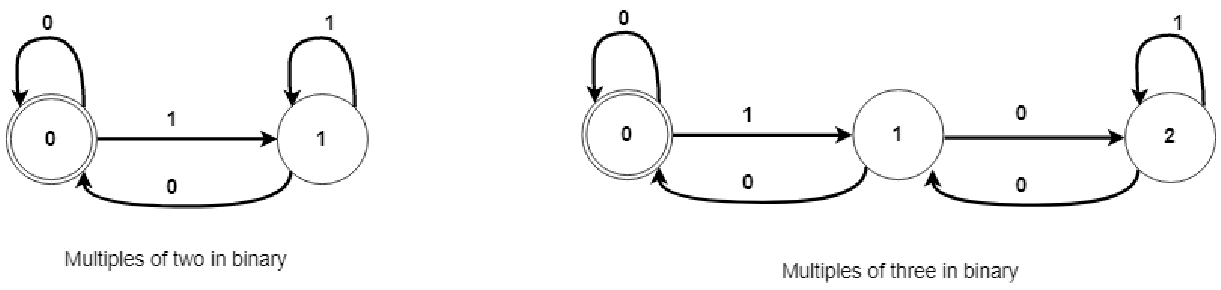


Figure 2.7: Binary multiples of two and three as FSA [78, p. 4]

2.5.4 Dynamic analysis techniques

Certain techniques are unique to dynamic analysis which is done testing or analyzing the program's behavior during run-time. The techniques may be done either in black-box or white-box analysis, though black-box is more common.

Dynamic Information Flow Tracking (DIFT) is a run-time technique for tracking information during the execution of a program. Information is tracked by tainting data, and piping the taint marks throughout execution [31].

Fuzzing is a program testing technique under the discipline of black box software testing. It consists of generating various semi-random data entries, malformed in order to test the input validation robustness of a system [67].

Crawlers are used to gather outside information from the front-end of a website. They can validate URLs and HTML code, and can traverse through and scan the application's pages. While they are most often used for indexing in search engines, their analysis capabilities work well for black-box program analysis [12].

A Web proxy, or a proxy server, acts as a middle-man for requests between clients and servers. Proxies are used for security, and their primary use is for providing access to the internet from within a firewall. The proxy can do efficient caching of all clients connecting to the server [53]. The information gathered from the HTTP communication that the proxy handles, can be used to perform security analysis.

2.6 Validation of program analysis tools

2.6.1 Validation strategies

Shaw [82] presents an informative analysis of what kind of research strategies are considered to be excellent in software engineering. Research deliveries in software engineering are classified by their types of research questions, results and validation strategy they choose. What kind of questions are interesting, and what kind of results can answer these questions, and what kind of evidence demonstrates the validity of the results?

The first part is choice of questions. We can choose from pragmatic questions like *method of development* on the form “How can we (better) create X” to analysis methods or design of particular instances. In the other end of the scale we have generalization, characterization or feasibility questions, more on the form “What is a good model for X” or “Does X even exist?”. The more common kind of questions in international conferences tend to be of an improved method or means of developing software, and analysis methods or testing and verification questions.

The second component in this is the research result. The various kinds of results range from procedure or technique, to qualitative, empirical or analytical models, tools (formal language to support a model), specific solutions, judgments or reports of interesting observations. On one side with the models, we often look at formal results like taxonomies or data-driven models. The work is highly constrained and often based on long and rigorous data collection. On the other end with Specific solutions, judgments or reports, we are looking at pragmatic software engineering solutions applied to problems, or careful analyses of a system. The nature of the results in combination with the design of their evaluation tells a lot about the validity of the research results.

A typical tendency has been that too many computer science papers contained no experimental or only informal validation of their contributions. The various choices for validation strategies differ in the value they contribute, ranging from a *blatant assertion*, which is no serious evaluation of the results, to analysis, which is a thorough an time demanding task. The choice of this strategy will impact the strength of evidence that the results of the research are in fact sound. The two most commonly accepted methods are experience in actual use and systematic analysis. However well-chosen *slice of life* examples rooted in reality are more convincing than idealized dummy examples, and reported as a common method to use. Also Oates [65, pp.115.118] argues for the importance of real-life validation to get convincing results.

2.6.2 Validation metrics

The metrics for validating program analysis tools or other classification problems often take basis in computing false and true positives, and false and true negatives. Here these metrics are defined in the context of program analysis tools reporting vulnerabilities.

- **True positives** (FP) are cases where the analysis reports a vulnerability, and this vulnerability exists.
- **False positives** (FN) are cases where the analysis reports a vulnerability, but no such vulnerability exists.
- **True negatives** (TN) are cases where there is no vulnerability, and the analysis does not report any vulnerability.
- **False negatives** (FN) are cases where existing vulnerabilities that don’t get reported.

This can be illustrated in a *confusion matrix*:

Table 2.3: Confusion matrix illustrating cases of true and false positives, and true and false negatives.

	Vulnerability exists	**No vulnerability exists**
Analysis reports vulnerability	True positive	False positive
Analysis reports no vulnerability	False negative	True negative

Soundness and completeness are properties that are normally used to quantify an analysis tools. Metrics for the properties come from computing false positives and false negatives. Soundness and completeness have various definitions, but a commonly used definition is as follows [26]:

The soundness of a program analyzer denotes whether it reports report all the issues in the code. A sound analyzer may have *false positives*, but reports all existing issues (meaning no false negatives).

The completeness of a program analyzer denotes whether it only report true issues. A complete analyzer may have *false negatives*, but all its reports are true (meaning no false positives).

It is however pointed out by Meyer [56] that soundness and completeness are boolean properties in the sense that, either a tool is sound, or it is not. However in the assessment of tools it is more interesting to look at the degree to which it can achieve one of these properties. To get a more granular sense in evaluating the metrics of a tool, the properties *precision* and *recall* are often used instead to define the degree of completeness and soundness.

- **The recall** of a program analyzer is the percentage of the existing vulnerabilities that are detected, in other words *how sound* is the analysis.

$$Recall = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **The precision** of a program analyzer is the percentage of its reports which are true cases (in other words the true positive rate). We can say that the precision denotes the degree of *how complete* the analysis is.

$$Precision = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

A factor for both the precision and recall is that they use data from both columns of the confusion matrix. This makes them sensitive to changes in data distributions, and may give a skewed perspective on imbalanced data [90]. If two data sets have different numbers of positives, the data distribution quickly changes. There exist metrics than can account for imbalance of the data.

To deal with this, the *true negative rate* (or specificity) is an interesting metric, which denotes a classification model's ability to correctly predict true negatives. This gives another perspective on how it resists false positives, without having to have a large volume of positives (which is required to get a confident precision) [1]. Both the identifiers in the metric lie in the same column of the confusion matrix, so it is not sensitive to imbalanced data that easily occur in small data sets [90]. This is because the changed values cancel each other out.

$$\textit{True negative rate} = \frac{\text{True negatives}}{\text{False positives} + \text{True negatives}}$$

Additionally, it is common in research on static analysis to calculate the *imprecision* of a tool, by calculating the *false positive rate* (FPR) [2]. This metric is the proportion of the positive warnings that are false positives, and gives a more direct sense of how much noise the analysis has, instead of its absence of noise. False positive rate (FPR) would in this sense be $FDR = 1 - P$ where P is the precision.

$$\textit{False positive rate} = \frac{\text{False positives}}{\text{False positives} + \text{True positives}}$$

There is a clear trade-off when designing a program analysis tool for soundness or completeness, as aiming for one of them will limit the capability in the other in a real-world domain, and no program analysis tool fulfills both criteria. Designers of program analysis tools must reason about whether they want to sacrifice completeness (precision) or soundness (recall) [30].

2.6.3 Choice of test-beds affecting metrics

Seng et al." [81] looked into how the quality of "security scanners" (or program analysis tools) is quantified. An common way is to challenge its features with various targeted test-beds. A test-bed may be a deliberately vulnerable web application with a known finite set of vulnerabilities in *artificial code*, or any other given code base with *natural code*. They found that 45.6 % of experiments use open-source web application frameworks, while fewer use educational sites or targeted test sites. The choices for test-beds range from custom-made web applications to educational vulnerable applications, test sites, open-source applications and real-world applications. These test-beds have different trade-offs that must be considered while quantifying an analysis tool. Deepa and Thilagam [24] show an overview of some test that are used for evaluating scanners or prototypes. All the commonly used test-beds in their case were open-source applications. The applications vary from content management to e-commerce and image managements systems.

Common metrics for evaluating an analysis tool are vulnerability detection rates with precision and recall, false positive or false negative rates, as well as scanning time and au-

[1]The true negative rate is also known as the specificity, which is commonly used in medical research to quantify a testing procedure.

[2]FPR here means the same as what the *false discovery rate* (FDR) often does in statistical classification problems. Several publications related to program analysis [13, 48, 98] however, refer to False Positive Rate with the definition given in this thesis.

tomation level. Such metrics are more convincingly computed in a deliberate test-bed, because it is hard to know about all the vulnerabilities in a real-life code base. On the other side, artificial test-beds risk being too far from real code-bases in terms of complexity and to which degree they cover realistic cases.

Three ideal test case characteristics are realism, statistical significance and ground truth. Natural code bases offer realism, and may also provide statistical significance if in large enough volume. However they lack ground-truth (we don't have knowledge of all their vulnerabilities).

Delaitre et al. [26] found the following metric applicability for real-life natural test beds versus artificial test-beds:

- For *natural* test-beds, precision is applicable, while it is hard to get convincing recall rates because of the ground truth problem.
- For *artificial* test-beds, both precision and recall are applicable metrics.

2.7 Architecture of the FindSecBugs plugin

Find Security Bugs [74], or often referred to as just FindSecBugs, is a security framework built as an extension of SpotBugs [84, 93]. SpotBugs is the successor of the original open-source FindBugs [1, 44] framework, which was developed with the concept of using simple static analysis techniques to identify bugs based on certain bug patterns. The idea is that it will pick up low-hanging fruits based on mistakes that may easily occur. SpotBugs has therefore inherited the original core engine of the framework, and is build and modernized around the original concepts. FindBugs and SpotBugs will hereafter be referred to around each other as the same.

The overall conceptual model of the FindSecBugs extension to the SpotBugs is illustrated in Figure 2.8.

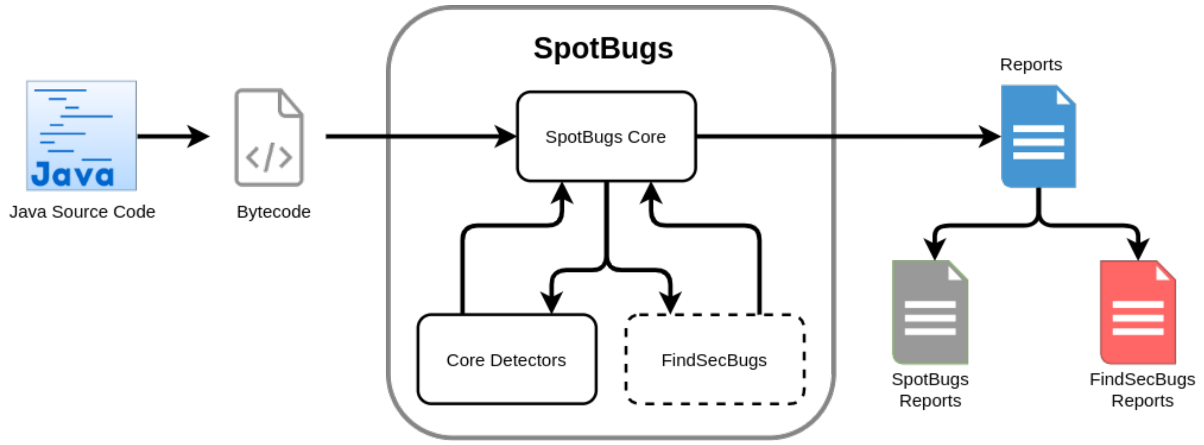


Figure 2.8: FindSecBugs is integrated into the SpotBugs framework, utilizing its core detectors [72].

The basic structure of FindSecBugs is built on two main components [69]:

- **Bug**: The definition of a sensible point or a vulnerability in the application. The definition of a bug exist by its presence in the project configuration files `findbugs.xml` (which corresponds detectors to bug patterns) and `messages.xml` (which contains descriptions of the bugs and suggestions for fixes). When the bug is defined in these files it can then be reported by detectors.
- **Detector**: A class containing the logic to find a bug type or a set of bug types. In other static analysis tools it is also common to refer to these as “rules”.

The building blocks for FindSecBugs lies in the core framework, SpotBugs, which in its order is built using Java Virtual Machine (JVM) bytecode abstractions from the Apache Commons Bytecode Engineering Library (BCEL) [3]. Compiled Java code which is interpreted by the Java Virtual Machine is located in .class files. FindBugs is designed to analyzed these files.

2.7.1 The SpotBugs Core Framework

The various detector types that exist can be divided into several layers of sophistication, depending on the bug pattern that is to be found. The detection strategies in the SpotBugs detectors can be viewed in four layers as shown in Table 2.4. In implementing the Detector interface, the `visitClassContext()` method is specified. The method takes an instance of `ClassContext`. which serves as a cache for the results from an analyzed class, as certain kinds of analysis is shared by many detectors. By collecting these results an increased memory cost is taken for a reduced CPU time.

[3]Apache commons BCEL: `https://commons.apache.org/proper/commons-bcel/`

Table 2.4: Overview of the layers of the various detector types in the SpotBugs framework as shown in the FindBugs paper by Hovemeyer and Pugh [44].

Layer	Description	Example Detector
1. Class structure and inheritance	Only the structure of the analyzed classes is inspected, without directly looking at the code.	The detector for "Useless Subclass Method", which only reasons about the structure.
2. Linear code scan	The bytecode is scanned linearly, analyzing each of the methods. The visited instructions are used to power a state machine. While not making use of complete control flow information, these use heuristics to effectively approximate control flow.	The detector for cross-site scripting reasons about several key-hole properties to find vulnerabilities in HTTP communications.
3. Control sensitive analysis	An accurate control flow graph is constructed for the analyzed methods.	The detector Check-TypeQualifiers
4. Dataflow analysis	Both control flow and data flow are taken into account, making for a deeper understanding of the program's properties.	Taint analysis detectors for SQL injection, and the detectors for unreleased locks.

However for relevance these layers are mainly divided into two rough categories, visitor-based detectors (layers 1 and 2), and CFG-based detectors(layers 3 and 4), as elaborated in an architecture document [4] written at the time of version 0.94 of FindBugs by the project founder, David Hovermeyer [23]. The visitor-based detectors are usually based on peephole techniques, and are very computationally lightweight. The CFG-based detectors doing control-flow and dataflow analysis are often heavier to run as they require more memory with graph-based operations. Therefore if a peephole check is sufficient to quite confidently classify a bug, usage of CFG-based analyses should be considered carefully. Despite this distinction, the SpotBugs framework lays no real constraint on the way a detector is implemented, and "any" analysis technique may be incorporated into a detector. In the end of the day however, a bug detector has a very straightforward task: look at a compiled java class file and find potential bugs, reporting them by creating a `BugInstance` object reporting it via the `BugReporter`.

Visitor-based detectors

These visitor-based detectors often extend the class `OpcodeStackDetector`, which ultimately is a subclass of `DismantleBytecode`. The basis behavior of these detector types is a top-down traversal of the class file's features, decoding the symbolic information. When the de-

[4]Due to the document being dated on certain points, small modifications and corrections are made, taken from code investigation of the current github repository of SpotBugs [84].

tector encounters a feature like a field, instruction, method or others, a callback method is invoked by the super class.

Visitor-based detectors can inspect the class file for suspicious features by overriding these callback methods. In visitor-based detectors, a state machine (See Section 2.5.3) recognizer that works over the sequence of instructions is introduced as an important idiom. The method `sawOpCode()` is the callback method handling individual instructions. Every invocation of this method is a single input symbol to the state machine. The state machine is practically a finite state automaton accepting a regular language. This language is a pattern that indicates trouble if it appears in the bytecode for a method. This method is quite simple in its management with control flow, but turns out to be significantly faster than the CFG-based analyses.

The role of the `OpcodeStack` class, which is present in the subclass `OpcodeStackDetector`, is to maintain information about the operand stack as the instructions in a method are visited, however still in a rather unsophisticated manner. While SpotBugs does not use context-sensitive, inter-procedural analysis, some detectors reason about global information like fields access throughout the application or sub-type relationships [8].

CFG-based detectors

Detectors employing the second layer with linear code scans are widely used both in the SpotBugs and the extended FindSecBugs framework. Some of the more advanced analyses developed in the SpotBugs core that do control- and data flow flow analysis can be utilized by FindSecBugs detectors. In this analysis, a CFG representation is built from Java methods, and the detectors usually implement the Detector interface directly instead of inheriting from a visitor superclass.

The fundamental behavior of the analysis is to sequentially visit each method of an analyzed class, requesting a set of analysis objects, which are end products of a certain analysis. An analysis object records certain and probable facts about the method based on for instance a dataflow analysis. After collecting these analyses, the detector iterates through each location in the control flow graph. Here a location is the point in execution just next to where a certain instruction is to be executed.

The dataflow facts are checked at every location for suspicious heuristics. For instance, the `ResourceTrackingDetector` class [5] is an abstract analysis class designed to find methods in which a resource of a kind is not properly cleaned up or closed properly. In this analysis the instructions creating an object are expected not to have a path through the CFG which does not lead to a close. Such a path will be considered an "open" path, and the method will be reported as a bug. One case in which this class can be extended is when database connections have not been properly closed in a finally block, and an exceptional control flow may therefore lead to an unclosed connection.

[5]ResourceTrackingDetector in the SpotBugs project https://github.com/spotbugs/spotbugs/blob/master/spotbugs/src/main/java/edu/umd/cs/findbugs/ResourceTrackingDetector.java

2.7.2 Detectors in FindSecBugs

The development of new detectors follows a defined work flow inspired by test-driven development [71]:

1. A vulnerable test code sample is added to illustrate the bug. As it is only intended to trigger the rule defined in the detector, it does not have to be a working application.
2. Then a test case is written, asserting that the given bug pattern was reported in the expected code location by the detector.
3. The new detector is configured by adding a detector and bug pattern to `findbugs.xml`.
4. Descriptions of the bugs and suggested fixes are added.
5. Finally a Detector is written, reporting the expected annotated bug patterns.

There are several detector classes that can be extended depending on the characteristics of the bug pattern that is searched for. The main types are:

OpcodeStackDetector searching for a specific method call,

ConfiguredBasicInjectionDetector searching for injection-like vulnerabilities, and

Detector which is the basic detector that analyzes the complete class context of a java class.

The simplest detectors use methods that are relatively easy to understand. Listing 2.5 shows an example of an XML related vulnerability, and Listing 2.6 has parts of its compiled bytecode. The bug is reported by the detector shown in Listing 2.7, which on line 10 specifically looks for the `invokevirtual` of the constructor of the XMLDecoder (line 5 in Listing 2.6).

```
public class XmlDecodeUtil {
    public static Object handleXml(InputStream in) {
        XMLDecoder d = new XMLDecoder(in);
        try {
            return d.readObject(); //Deserialization happen here
        }
        finally {
            d.close();
        }
    }
}
```

Listing 2.5: Vulnerable test code sample for usage of XML deseseralization.

```
1 public static java.lang.Object handleXml(java.io.InputStream);
2     Code:
3        0: new             #2 // class java/beans/XMLDecoder
4          ...
5        5: invokespecial #3 // Method java/beans/XMLDecoder."<init>":(
    Ljava/io/InputStream;)V
6          ...
```

```
      10: invokevirtual #4 // Method java/beans/XMLDecoder.readObject:()
   Ljava/lang/Object;
        ...
      28: aload          4
      30: athrow
      ...
```

Listing 2.6: Bytecode of compiled vulnerable test code sample for usage of XML deseseralization.

```java
public class XmlDecoderDetector extends OpcodeStackDetector {
    private static final String XML_DECODER = "XML_DECODER";
    private static final InvokeMatcherBuilder XML_DECODER_CONSTRUCTOR =
   invokeInstruction().atClass("java/beans/XMLDecoder").atMethod("<init>
   ");
    private BugReporter bugReporter;
    public XmlDecoderDetector(BugReporter bugReporter) {
        this.bugReporter = bugReporter;
    }
    @Override
    public void sawOpcode(int seen) {
        if (seen == Const.INVOKESPECIAL && XML_DECODER_CONSTRUCTOR.
   matches(this)) {
            bugReporter.reportBug(new BugInstance(this, XML_DECODER,
   Priorities.HIGH_PRIORITY) //
                    .addClass(this).addMethod(this).addSourceLine(this))
   ;
        }
    }
}
```

Listing 2.7: Detector for usage of XML deseralization.

Find Security Bugs has also introduced a taint analysis component, which may be used by several detectors to track data between tainted sources and sinks. Several detectors in Find Security Bugs use resource files to list their vulnerable sources and sinks, which form inputs to the identifiers they want to check. Detectors read and use these lists when scanning the code. This enables the community to easily update the detectors without their logic, only their inputs. This may become useful when new vulnerabilities are discovered by the security community. All that is needed for detecting the new vulnerability may be to add a single line with a new identifier to the resource file. Such identifiers may also be used by other detectors than the ones directly applying taint analysis.

2.7.3 Vulnerability coverage and usability in FindSecBugs

FindSecBugs covers a variety of defined OWASP- and CWE-defined vulnerabilities in their detectors, as described extensively in an evaluation by Li, Beba and Karlsen [48]. However many vulnerability detectors are not based on very modern web-based security practices, and no coverage can be found directly for protocols like OAuth 2.0 and OpenID Connect. An exception is the “Hard-coded password detector” observed in the source code of the plugin, which has been extended to detect usage of hard-coded client IDs and secrets in a Spring OIDC configuration.

FindSecBugs has inherited the usability of FindBugs, and offers integration in IDE or other development phases like continuous integration steps. Users of the plugin may suppress false positives and target their analysis towards certain packages or classes. When going through warnings, it is likely that it is quick and easy to fix, and only requires inspection of a few lines of code [8, 74].

Chapter 3

Related Work

This chapter goes through the related work to this thesis related to the two research questions. The first part 3.1 goes through formal security analyses and models of the protocol, which relate to RQ1. Section 3.2 summarizes automated vulnerability detection and protection tools for the protocol, which are relevant to RQ2.

Related research was obtained with informal searches for relevant keywords like *OpenID Connect, vulnerabilities, detect, static/program analysis*, in Oria (The digital library at NTNU), Google Scholar, the ACM digital library, the IEEE digital library and the relevant paper index in the Mendeley reference manager. The references and forward citing indexes of some of the paper were briefly scanned for more inclusions.

The previous research on OAuth and OpenID Connect is often focused on formal security analysis and threat modeling. The research works can be considered in to factions; One uses formal security analysis based in threat models of the specification, or manual analysis of implementations to reason about security vulnerabilities in the specification. The other faction uses automated penetration testing or program analysis tools to find vulnerabilities in implemented OpenID Connect systems.

3.1 Security analysis of OAuth and OpenID Connect Specification

This section highlight related work which relates to RQ1: *What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?*

The related formal security analyses are generally using two different approaches. Some of the look at the specification itself, inferring vulnerabilities inherent to how the protocol standard is defined [33, 34, 62, 89]. Other formally analyze implementations of the protocol, and use results from experiments to find possible vulnerabilities [2, 49, 50, 86]

Sun and Beshnov(2012) [86] examined implementations of the much-used OAuth IdPs of Facebook, Google and Microsoft, as well as 96 facebook-connected RPs, using manual techniques to analyze HTTP messages. They wrote penetration tests to explore potential exploits. Their findings include the possibility that the confidentiality of access tokens can be broken, and the possibility to forge credentials and sending these to the sign-in endpoint of the RP.

In 2013, Lodderstedt et al. in developed a comprehensive threat model for OAuth 2.0 [89]. The threats they present with regard to Relying Parties revolve around an attacker's ability to obtain secrets, injecting malicious data or acting as a man in the middle. They also highlight cases where the developer may fail to properly protect sensitive data, or have errors in their configuration. An attacker impersonating an IdP may be able to obtain access to protected resources if the Relying Party is insufficiently protected against CSRF attacks. Usage of weaker grant types like Resource Owner Password Credential should be minimized as it eliminates the strength of the token-based flow.

Li and Mitchell(2014) [49] looked into security issues in 60 Oauth 2.0 implementations based in China, in case studies where they manually inspected the network communication. They found two critical vulnerabilities, and provide recommendations for how IdPs and RPs can mitigate these vulnerabilities. Their recommendations include that IdPs should take responsibility to include usage of the *state* parameter in the sample code in their developer guides. They should also give proper emphasis of the potential risks, to encourage RP developers to include this value.

Fett et al. conducted two studies looking at the security properties of OAuth 2.0 in 2016 [33] and of OpenID Connect in 2017 [34]. They analyzed the protocols in an abstract manner, mostly ignoring implementation details. An extraction of the vulnerabilities they presented are shown in Table 3.1, vulnerabilities 1-10.

Alaca and van Oorschot (2018) [2] analyzed and compared 14 Web SSO systems, including Oauth 2.0 and OpenID Connect. They developed a taxonomy for SSO schemes by identifying common design properties. They discussed how priorities related to users, RPs and IdPs impact how SSO schemes are designed and deployed. They highlighted how different schemes provide benefits for different use cases.

Li, Mitchell and Chen (2018) [50] continued their work from 2014[49], and looked deeper into how CSRF attacks can be mitigated in OpenID Connect as well as OAuth 2.0. They suggest including more sophisticated usage of *referer headers* in addition to the protocol-specified *state* parameter to mitigate CSRF attacks against the *redirect_uri*, by tracking the intention of the user. This mitigation is related to the findings of Fett et al. [34]. The threat is shown as threat number 11 in Table 3.1.

In 2019, Navas and Beltrán [62] did a thorough treat modeling of the OIDC core specification, and some of its implementations. In total they identified and described over 16 different attack patterns, as well as other threats to privacy and security. Their main contribution that separates them from other works is their *crafted token attack*, which is described as threat 12 in Table 3.1.

Sadqi et al [80] analyzed the security properties of Single sign-on systems in 2020. They present security implications of different protocol flows in OAuth 2.0, and an examination of security issues related to both the protocol specification and its implementation on the Web. They focused on how the security parameters like state values and tokens play roles in the security of the protocol, and how these values can be exposed. One of the threats they highlighted is usage of the *bearer token*, since this does not offer data origin validation in itself. They suggest that signature-based access tokens should be used instead. This problem is however solved by OpenID Connect, using ID tokens. Additionally such signature-based tokens are described in OpenID Connect as an optional part of the standard [60], for when using the implicit flow.

3.1.1 Threats on OpenID Connect Relying Parties

When looking exclusively at the Relying Parties of the protocol, there are still many different attack surfaces. This section summarizes the threats and vulnerabilities that have previously been found the *client* or *Relying Party* (RP), which is especially vulnerable entity because it is usually the one written by the non-security specialized developer.

Table 3.1 shows an overview of the threats on OpenID Connect Relying Parties identified by related work. Threats that do not directly involve RPs are intentionally left out.

Table 3.1: Overview of threats on OIDC Relying Parties [33, 34, 50, 62].

Threats on RP	Vulnerability	Mitigation
1. IdP mix-up attack [33, 34]	1. RP is confused and sends credentials to attacker's IdP.	1. IdP should put its identity into the response, RP verifies the identity. Applies only to Implicit Flow.
2. Attack on state parameter (CSRF) [33, 34]	2. The same state parameter is used, and can be replayed.	2. Nonce for the *state* is chosen freshly on login bound to user's session.
3. Code/Token/State leakage [33, 34]	3. State is leaked through referrer header to third-party through script/link.	3. Documents delivered at endpoints should be vetted for links to external resources.
4. Naïve RP session integrity attack [33, 34]	4. RP puts the IdP identity into the redirect URI (Naïve user intention tracking).	4. RP should always use sessions to store user's chosen IdP (Explicit tracking).
5. Injection attacks [33, 34]	5. Proper escaping of data parameters in the redirection URIs is lacking.	5. Vetting all input data from untrusted sources, carefully escape output data.
6. CSRF and third-party login initiation [33, 34]	6. A third-party can initiate login by redirecting user to Login initiation endpoint. This is an optional feature bypassing the state-based CSRF protection.	6. Login initiation endpoints should not be implemented.
7. Server Side Request Forgery [34, 68] : the attacker manipulates the redirect URI.	7. RP is instructed to send requests to a malicious discovery party.	7. Proper filtering and mechanisms to limit server-based requests.
8. Third party resources [33, 34]	8. Untrusted third-party resources embedded in the same origins as RP endpoints.	8. Avoid embedding third-party resources, or demand integrity through a specific hash match.
9. TLS security [33, 34]	9. Data is transmitted in cleartext and credentials are exposed.	9. Parties in OIDC should use and require HTTPS in URIs.
10. Bad Session Handling: session fixation attacks [33, 34]	10. The RP forgets to replace the session id with a fresh nonce.	10. The RP should always use fresh nonces after login, and store nonces in a cookie with the "Secure" property.
11. CSRF [50]	11.Attacker intercepts redirect, modifies $redirect\_uri$.	11. Use referrer header to track intention.
12. Crafted tokens [62]	12. Attacker exploits incomplete token validation at the RP.	12. Properly validate all the values in the ID token.

3.2 Automated tools for detecting OpenID Connect vulnerabilities

This section goes through research which is relevant for RQ2: *How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?*

Previous research have made both dynamic- and static analysis solutions for security verification of OpenID Connect and OAuth, with various focus areas and abstractions.

Wang et al [95] performed an analysis of three authorization and authentication SDKs in 2013, which at the time were used by 52% of the most popular Windows App store apps. They used a semantic modeling-based approach using knowledge bases to explicate the SDKs. The approach generates formalized assertions that are checked based on a clause of semantic model properties, and detects security violations by testing proofs with a satisfiability problem (SMT) solver. Using a symbolic execution framework for validation of the models, analyses took between 11 and 25 hours to check the three SDKs for vulnerabilities.

In 2014, Zhou and Evans [99] introduced SSOScan, a black-box penetration testing tool for Relying Parties, applications using SSO. They conducted a large-scale study, which is limited to RPs using Facebook's implementation of OAuth. They detect four vulnerabilities: access token misuse, app secret leak, user OAuth credentials leak and signed request misuse. The former two are related to confusion regarding authorization mechanisms, while the latter two are based on failures to keep secrets confidential. SSOScan simulates a series of attacks and observes the responses that come over the network. The tool has an regex-based automated button finder on the forms in the sites that it analyzes. The tool is limited to faking user interactions and as a black-box tool limited to vulnerabilities can be detected through analyzing web traffic patterns.

Yang et al.(2016) [97] designed and implemented a model-based tool called OAuthTester. They examined found major identity providers and 500 websites implementing OAuth 2.0. In their design they use a finite State machine to model the protocol flow. They use fuzzing (See Chapter 2.5.4) techniques to query the RP and the IdP. They mainly found vulnerabilities related to improper management of the *state* parameter.

Mainka and Wich [54] proposed in 2017 an Evaluation-as-a-Service tool they call PrOfESSOS, which dynamically allows a tester to perform black-box penetration testing in run-time, simulating honest and dishonest IdPs. They categorize to main classes of threats; Single-Phase Attacks (exploit a single security check) and Cross-Phase attacks (complex attack setup manipulating several messages in the data flow). These classes encapsulate most of the various threats summarized in Table 3.1. HTTP requests are manipulated by the tool's IdP, and RP reactions to different malicious requests are analyzed. Detection criteria for a vulnerability is determined by successful maliciously obtained access to credentials. The analysis requires a manual configuration to increase soundness.

Yang et al. (2018) [96] designed an automated testing tool, S3KVetter, verifying logical correctness and identifying vulnerabilities in SDKs implementing OpenId Connect or OAuth.

Their focus is in SDKs that are used for implementing a client application, as a more specific continuation of their previous work [97]. Their approach is based on *theorem provers* after the program's code is translated to appropriate logic predicates. The code is by dynamic symbolic execution extracted to a symbolic predicate tree, in which all the program's execution paths form branches in the thee, with the leaf nodes containing the end result of a given path. The paths are explored with a scheduling algorithm that simulates various program executions with data inputs. As their approach is focused around attacker-oriented steps of the protocol flow, their analysis cannot reach different paths than an attacker might, and their knowledge of the program internals is lacking. Their notion of an attacker is a malicious "user" doing man-in-the-middle attacks. Hence their approach also assumes that the IdP is trustworthy. The tool requires some manual setup of a sample app, and the user must mark which functions may be reached by an attacker (functions handling user input).

Calzavara et al. (2018)[11] made a browser-side security monitor called WPSE, and a thorough security analysis of Web protocols, including OAuth 2.0. Their tool is designed to ensure compliance with the intended protocol flow, and integrity and confidentiality of messages. In an experiment on 90 websites, they uncovered that over 61% had security flaws. This browser extension must presumably be installed by the website's users.

Li, Mitchell and Chen. (2019)[51] proposed at roughly the same time a security scanner and protector, OAuthguard, which similar to WPSE provides protection for OIDC and OAuth 2.0 as a browser extension. They performed an experiment on the top 1000 RPs using the Google single sign-on services as IdP. Like other dynamic analyses, this tool acts as a proxy, and detects vulnerabilities by scanning HTTP messages. It may block http requests if the request indicates unsafe token transfer (checking TLS usage), privacy leaks, impersonation attacks and CSRF attacks.

Also in 2019, Rahat et al. [76] introduced OAuthLint, a tool using query based static analysis to find vulnerabilities in Android apps that implement the protocol using OAuth APIs. They based their analysis on a model with *anti-protocols*, which denotes vulnerabilities in the protocol. They analyze relying parties for vulnerabilities related to:

- Local storage of critical values like tokens,
- hard-coded client secrets,
- improper encryption of access tokens data in transmission,
- usage of an insecure protocol called WebView for data transactions and
- failed validation of API calls from client Android devices which should not be trusted.

Their analysis computes a control-flow graph which they query with formal logic predicates. They evaluated their analysis on around 600 popular Android apps, and found that 32% of the analyzed apps had at least one of the five vulnerabilities they looked for. Their analysis achieved a high precision of 90%.

Table 3.2: Overview of related works doing automated vulnerability detection of OpenID Connect and OAuth 2.0.

Publication		Type of analysis	Description	Protocol
Wang et al. (2013)	[95]	White-box	Symbolic execution	OAuth 2.0
SSOScan (2014)	[99]	Black-box	Penetration testing	OAuth 2.0
OAuthTester (2016)	[97]	Black-box	Fuzzing	OAuth 2.0
PrOfESSOS (2017)	[54]	Black-box	Penetration testing	OIDC
S3KVetter (2018)	[96]	White-box	Symbolic execution	Both
WPSE (2018)	[11]	Black-box	Browser security monitor	OAuth 2.0
OAuthguard (2019)	[51]	Black-box	Browser monitor and proxy intercepting request	Both
OAuthLint (2019)	[76]	White-box	Static analysis	OAuth 2.0

3.3 Precursory thesis work

Preceding this thesis, I conducted a review [87] of relevant research in program analysis tools for detecting access control vulnerabilities [10, 22, 25, 27, 32, 43, 46, 47, 57, 58, 59, 63, 77, 79, 83, 85, 98, 100, 101]. These provide various methods for vulnerability detection with regards to access control. Many of the methods are designed for a more traditional client-server based access control model, and are not directly suitable for analyzing applications using OpenID Connect. Additionally, the precursory work includes a published research paper written in cooperation the supervisors of this thesis [88]. This work forms an important basis for the ideas of this project (The paper is also attached in Appendix A). Our study on software consultants with 80 respondents had the following key results:

- Consultants have a near 50/50 distribution in preference that the static analysis tool is fully automated (and is therefore less precise but easier to use) or requires some annotations but more powerful.
- If the tool seamlessly integrates into their workflow, software consultants are more likely to use it.
- The respondents answered that they generally do not think precision should be lower than 90%.
- However, lower precision is acceptable if security code is analyzed. We found in our study that the consultants are much more inclined towards in higher recall (or soundness) than high precision if the tool looks for security-critical vulnerabilities like access

control vulnerabilities.

Chapter 4

Research Design

4.1 Research motivation

OpenID Connect (OIDC) is becoming increasingly common in modern Java applications as a de facto standard for authentication and authorization with Single sign-on federation services. Vulnerabilities in OpenID Connect can be considered a subset of *Access Control Vulnerabilities*, the fifth highest ranking weaknesses according to the OWASP top 10 list of Web Application Security Risks [92].

There have been several known cases of data breaches due to insecure Single Sign-On implementations in the later years, like the Facebook breach in 2018 [55], where millions of access tokens were hijacked. Due to insecurely implemented clients lacking proper session management, adversaries could gain access to hundreds of websites outside of Facebook itself, with no way for the users to revoke the attacker's access. Developers may use well-known SDKs for building a Relying Party (RP) in OIDC to connect their app with an Identity Provider (IdP) for identity management. Examples of such SDKs are the Nimbus OAuth SDK [18] and Google OAuth Client Library [38, 39].

Even if these SDKs help the developer by encapsulating several difficult implementation details, the developer and the SDK still share a common responsibility in securing the RP application. The SDKs give tools for managing Web-specific features, and can provide strong data types for the data delivered between the RP and the IdP. Still, the developer is responsible for establishing a trust relationship with the IdP, and for correctly managing secrets and data that are needed to ensure integrity, confidentiality, and non-repudiation in the communication. Even though existing solutions have been made to automatically detect vulnerabilities, more must be done since several clients on the Web still have vulnerabilities in their production code.

A large portion of the implemented protocol steps in OpenID Connect clients are likely to share similar (uncomplicated) structural and syntactic properties. These are easy to check with static analysis. Therefore simple analyses may be enough to mitigate a substantial part of the vulnerabilities that can come in OpenID Connect clients.

4.2 Research questions

This thesis seeks to answer the following research questions:

RQ1 What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?

RQ2 How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?

4.3 Research strategy

The research strategy in this thesis is formed based on guidelines from a paper by Shaw [82] and from the book by Oates [65]. For the purpose of clarity, the terms from Shaw's paper are highlighted in defining the research strategy.

4.3.1 RQ1

In this thesis, RQ1 falls into the *Characterization* category. This means that the desired results are a form of document, list or model. The results from RQ1 are a qualitative model in form of a well-grounded checklist and informal generalizations. The results from RQ1 are considered input to the work in RQ2, and the validation of these results is therefore done indirectly through the validation of the results emerging from RQ2.

4.3.2 RQ2

RQ2 is in the *Design of a particular instance* category for research questions. It is not seeking a formal model or any general framework, but rather a pragmatic and concrete solutions to a concrete problem. The validation strategy chosen in this thesis can be characterized under the *Example*-based category, which implies some threats to the validity of the results (This is discussed further in Chapter 8.3). While this is not considered as strong as the more ideal choices, of *Analysis*-based or *Experience*-based validation strategies, well-chosen *slice of life* examples can be considered somewhat successful and is fairly common in Software Engineering research [82].

It was was considered too time-consuming for the constraints of this work to design a statistically significant empirical validation of the results in this thesis, due to the work load required to obtain a sufficient volume of code bases or corpus, as well as a statistically rigid design for the experiment. Therefore a simpler alternative with slices of life were considered an acceptable plan B. As such, this research is mainly answered through what Oates [65, pp.133–134] defines as *field experiments* on real-life code examples. The example-based strategy can also according to Oates be considered a *proof-by-demonstration*.

4.4 Data generation and analysis

Observation [65] is considered the main data generation method in this thesis.

The data generated to answer RQ1 is based on collating documents produced by the protocol specification authors, security researchers, and the code and belonging documentation of the chosen SDKs. Data from these documents were analyzed qualitatively.

To answer RQ2, the data generated are results from a field experiment running the analyses on real code, with additional manual code inspection to verify and summarize the results. These data are also mainly analyzed qualitatively, as the volume of code examples would not give statistically significant numbers if analyzed quantitatively. The quantitative metrics of *precision*, *recall* and *true negative rate* are included for some reference. However there cannot be statistical confidence in them, and they serve more of an illustrative purpose rather than an absolute metric for a strict comparison. These metrics are not statistically significant, both because of lacking volume of code material, and because confident metrics requires a more controlled environment through a test-suite with known vulnerabilities. Precision and recall are sensitive to imbalanced data sets, while the true negative rate is considered more robust when facing imbalanced data [90].

The test suite was chosen mainly based on efficiency and realism. Because of time constraints, six open-source applications were included in the test suite, by first-and-best matching in searches. How these were obtained is explained in Chapter 7.1.

4.5 Design to answer RQ1: Analysis of OpenID Connect

4.5.1 Summarizing known vulnerabilities

The work with summarizing the common vulnerabilities goes along two axes: one is to summarize the identified vulnerabilities by research on vulnerability analysis of the protocols, with focus on the vulnerabilities that concern the Relying Party. However this alone is not sufficient to get a complete picture of the vulnerability domain, as the systems often are addressed form an abstract outside-perspective rather than on the code. Additionally for the other axis, security analysis is narrowed to look at implementation details on the RP client-side, with proposals for possible errors to make there. Other vulnerabilities which concern other entities in the protocol were out of scope. Related automated vulnerability detection tools were also looked at to get a view of the state of the art.

4.5.2 Analysis of the implementation details in RP

Two open-source SDKs with their developer guides were chosen as a basis for implementation details analysis: the Nimbus SDK by Connect2id [17, 18] and the Google API Client SDK [38, 39]. These guides contain developer instructions which likely put some design constraints on the development. Based on these stripped-down code examples and the OpenID

Connect Specification as well as research security analyses of the protocol (See Chapter 3), code examples were constructed with the intention of having as much realism as possible, without adding boilerplate code that can be considered out of scope for this thesis. These code examples form the fundamental building block for the pragmatic model for development as a means of an informational sweet-spot between the content-rich protocol, and the quite limited developer guides.

4.6 Design to answer RQ2: Implementation of static analyses

4.6.1 Selection of tool for basis of analyses

It was necessary to select an existing tool to implement and test the new techniques proposed to save configuration time in this project. The available software projects for preliminary testing and analyses were mainly written in java, another deciding factor when selecting tools for analysis. Li, Beba and Karlsen [48] did an experimental validation of various open-source IDE plugins that detect security vulnerabilities. They investigated both the vulnerability class scope, quality of detection and user-friendliness of the tool warnings. They found a mismatch between the claimed and actual coverage of the tools, and that high false positive rates were present. Several tools had limited information in their output, and the drawbacks ranged from imprecise or lacking explanations of the vulnerability itself, as well as missing educational value.

Another issue was missing opportunity to direct the tool, and some tools only had modes to scan the whole source code at once. Find Security Bugs [74] or FindSecBugs, an OWASP project, came out as the highest performing open-source tool in a total evaluation, with respect to usability and other quality metrics. FindSecBugs was therefore chosen as a basis for developing analyses in on a framework that is developer-friendly and prevalent in the developer community. Implementation of detectors took inspiration in how existing analyses in FindBugs and FindSecBugs are built, with the idea of following similar software design principles while adapting new analyses into the tool. The vulnerability detection strategy was based on the findings of the protocol model analysis that was done to answer RQ1. Details about the implementation is presented in Chapter 6.

4.6.2 Validation of the analyses

The “slice of life” example-based strategy (See Section 4.3), which also can be referred to as a *quasi-experiment* or field experiment, is used as basis to validate the results of this thesis. Further details of how the experimental validation was carried out is presented in Chapter 7.

Chapter 5

RQ1 results: developer-oriented model of secure OIDC practice

This chapter answers RQ1: *What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?*

The foundation of the developer-oriented model worked out in this chapter is restricted to the *Authorization Code Flow* of the protocol, with basis in two SDK implementation guides. The steps covered are metadata discovery, authorization code request, token request and token validation, which are shown as steps 0-3 in Figure 5.1.

5.1 A developer-oriented model of the OIDC flow

Many descriptions of the OpenID Connect flow include communications between user, RP and IdP, and often stay on a "bird perspective" with brief descriptions of what the RP does. These are seldom on a level of detail which is sufficient for developers to fully understand what is *their* role implementing the RP. This model is therefore brought onto the ground and into the implementation details of Relying Parties, looking at the protocol through the developer's eyes. Figure 5.1 shows a three-step flow that forms the core of the authorization code flow in OpenID Connect, divided into three different "main steps" for the RP developer, with a preparatory step 0, in which the *Discovery process*.

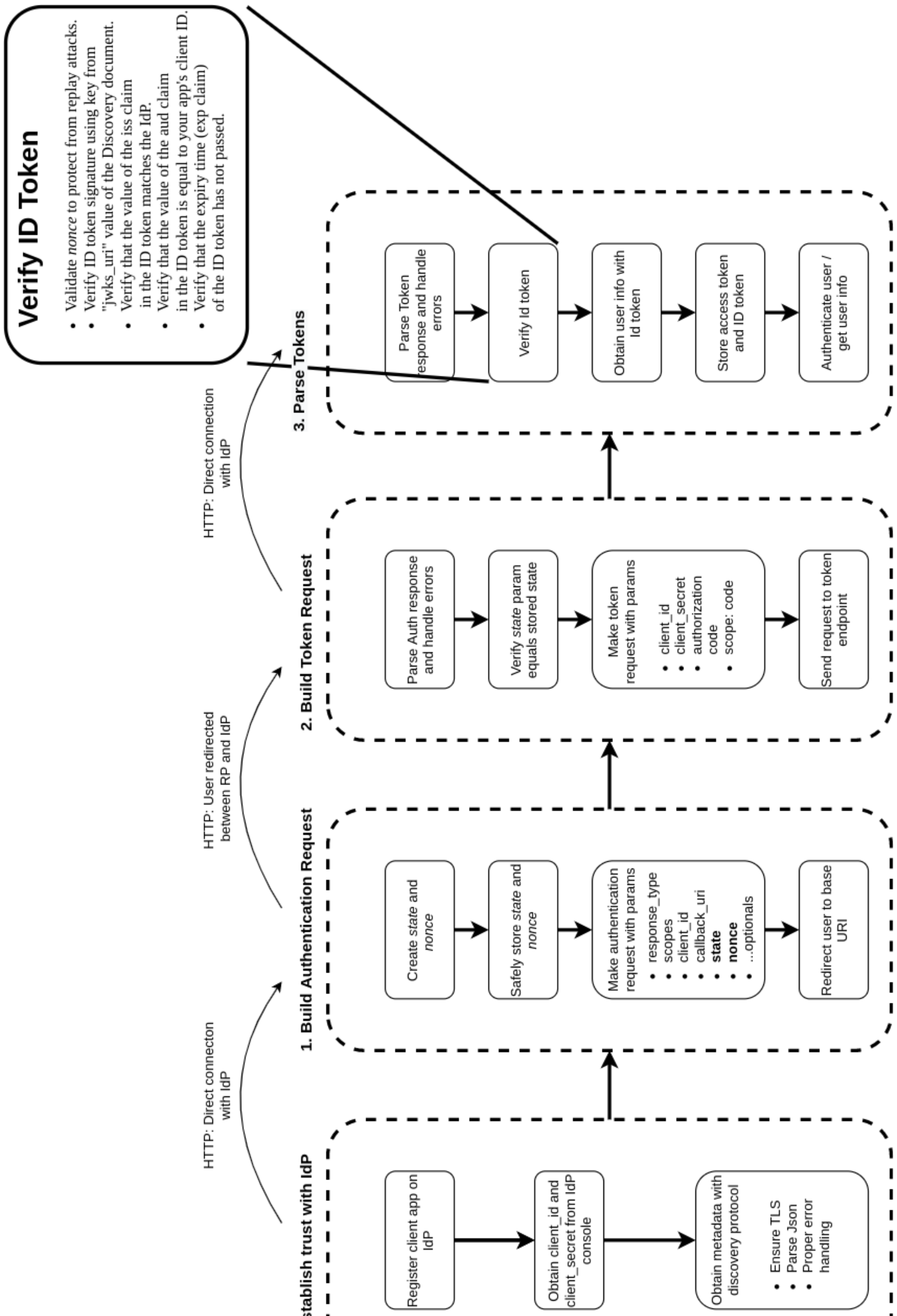

0. Establish trust with IdP
Register client app on IdP
Obtain client_id and client_secret from IdP console
Obtain metadata with discovery protocol
• Ensure TLS
• Parse Json
• Proper error handling
HTTP: Direct connecton with IdP
1. Build Authentication Request
Create *state* and *nonce*
Safely store *state* and *nonce*
Make authentication request with params
• response_type
• scopes
• client_id
• callback_uri
• **state**
• **nonce**
• ...optionals
Redirect user to base URI
HTTP: User redirected between RP and IdP
2. Build Token Request
Parse Auth response and handle errors
Verify *state* param equals stored state
Make token request with params
• client_id
• client_secret
• authorization code
• scope: code
Send request to token endpoint
HTTP: Direct connection with IdP
3. Parse Tokens
Parse Token response and handle errors
Verify Id token
Obtain user info with Id token
Store access token and ID token
Authenticate user / get user info
Verify ID Token
• Validate *nonce* to protect from replay attacks.
• Verify ID token signature using key from "jwks_uri" value of the Discovery document.
• Verify that the value of the iss claim in the ID token matches the IdP.
• Verify that the value of the aud claim in the ID token is equal to your app's client ID.
• Verify that the expiry time (exp claim) of the ID token has not passed.

Figure 5.1: A developer-oriented model of the authorization code flow in OIDC

These three steps (plus the preparatory step) are inferred from a sensible division of code into each their designated method. After step 3 naturally, the protocol would follow with a query to the UserInfo endpoint. This is however left out of this scope. Code examples with non-compiling Java-pseudo code of these steps are shown in listings below.

An example of the code in step 0, which is the Discovery process, is shown in Listing 5.1. This is a preparatory step in the model. Listing 5.2 has a code example of step 1, building the authentication request. The results from the pre-step 0 in the discovery protocol are obtained on line 4, where the RP has sent a request to the *discovery URI* of the IdP, and received the Provider Metadata Document in return to establish trust. The important things to remember here are adding the state and nonce parameters. The rest of the parameters are essentially required to even send a request.

Step 2 receiving callback response from the IdP is shown in Listing 5.3. Here handling an eventual error response and validating the *state* parameter are the security-critical steps.

Step 3 has the most significant difference for the two SDKs analyzed in this thesis, and is shown in two different ways using the Google library (Listing 5.4) and the Nimbus SDK(Listing 5.5). The main difference lies in that the Google library in Listing 5.4 does not have a completed validation encapsulated, and the developer must therefore handle details of the conditional checks.

Nimbus in Listing 5.5 the other hand, requires the developer to set up an `IDTokenValidator` object with some required parameters, and it will handle the individual checks and throw appropriate exceptions if something is unexpected. Here the developer still has to pass the correct values, however, and must properly pass the correct nonce value that they have stored.

In Listing 5.1, the Discovery process in implemented with the Google library. In lines 1-7, the RP builds the URL for the *openid-configuration* endpoint. Lines 10-16 contain checks to ensure that the connection is using TLS, and that the response is of a valid HTTP response code. Then the JSON document is retrieved in lines 17-21, and finally parsed. If the parsing fails, an appropriate exception is thrown.

```
1  // Step 0
2  private Map<String, Object> discovery() {
3      try {
4          URI issuerURI = new URI("https://provider.example.com/");
5          URL idpConfURL = issuerURI
6                          .resolve("/.well-known/openid-configuration?")
7                          .toURL();
8          HttpsURLConnection conn = idpConfURL.openConnection();
9          conn.setRequestMethod("GET");
10         if(!conn.getURL().getProtocol().equals("https")) {
11             throw Exception..."Discovery url not using https"
12         }
13         if(conn.getResponseCode() != HttpsURLConnection.HTTP_OK) {
14             throw Exception..."Failed to respond with HTTP OK."
15         }
16         InputStream stream = conn.getInputStream();
17         String providerInfo = "";
18         try (java.util.Scanner s = new java.util.Scanner(stream)) {
19             providerInfo = s.useDelimiter("\\A").next();
20         }
21         return parseJson(providerInfo);
```

```
    } catch (...Exception e) {
        throw Exception..."Failed to perform discovery"
    }
}
```

Listing 5.1: Step 0 - The Discovery process using the Google library

Listing 5.2 shows a simplified code example of step 1, the authentication request. First in line 4, the provider metadata document is obtained from the discovery process in step 0. The *state* and *nonce* values are generated from methods that make opaque randomized strings in line 5-6. These are stored in an object called OidcConfig. Then an *AuthorizationCodeFlow* object is built in lines 8-11, storing values obtained from the IdP. The client id and client secret have previously been obtained when registering the client at the IdP. Then in lines 12-19, the authentication request URL is build. The *state* and *nonce* parameters are added to the request. Then in lines 20-22, the requesting user agent is redirected to the authorization end-point at the IdP.

```
1  // Step 1
2   public Response authenticationRequest(HttpServletRequest request) {
3      try {
4          providerMetadata = discovery(); // Step 0
5          String state =  nonce(); // random string
6          String nonce =  state(); // random string
7          // ... Store state and nonce in OidcConfig
8          codeFlow = new AuthorizationCodeFlow.Builder(...,
9                  config.getProperty("clientSecret")),
10                 config.getProperty("clientId"),
11                 providerMetadata.get("authorization_endpoint")).build();
12         requestUrl = codeFlow
13                 .newAuthorizationUrl()
14                 .setResponseTypes(Collections.singleton("code"))
15                 .setScopes(scopes)
16                 .setRedirectUri(callbackURI)
17                 .setState(state)
18                 .set("nonce", nonce)
19                 .set(..., ...);
20         return Response
21                 .seeOther(requestUrl.toURI())
22                 .build();
23     } catch (...Exception e) {
24          return Response...UNAUTHORIZED...;
25     }
26 }
```

Listing 5.2: Step 1 - Authentication request using the Google library

Listing 5.3 contains a simplified code example of step 2, where the response from the IdP is received as a callback request. The *OidcConfig* for the given flow is retrieved with a unique UID in line 5. Then, the callback response URL is parsed. In lines 7-10, the callback response is checked for an error, and the flow is broken if it does have an error. Then comes an important check in line 11, where the *state* parameter in the callback request is compared to the stored value, and an appropriate HTTP error code is returned. After this validation, a token request is build in lines 14-19, adding the authorization code received in the callback request as well as other required parameters. Then the request is executed on a back-channel connection with the IdP in line 20. It returns a token response, and will throw and exception if this is not successful. Then the token response and the oidcConfig containing the *nonce* parameter are passed on to step 3.

```
1  // Step 2
2  public Response callback(HttpServletRequest req) {
3     try {
4          UUID uuid = UUID.fromString(...req.get(uuid));
5          OidcConfig oidcConfig = (OidcConfig)cache.get(uuid);
6          ...ResponseUrl responseUrl = new ...Url(req.getRequestURI());
7          String error = responseUrl.getError();
8          if(error != null) {
9              return Response...UNAUTHORIZED...;
10         }
11         if(!oidcConfig.state.equals(responseUrl.getState())) {
12              return Response...UNAUTHORIZED...;
13         }
14         String authorizationCode = responseUrl.getCode();
15         TokenRequest tokenRequest = codeFlow
16                 .newTokenRequest(authorizationCode)
17                 .setTokenServerUrl(codeFlow.getTokenServerEncodedUrl())
18                 .setClientAuth...(codeFlow.getClientAuthentication())
19                 .setRedirectUri(redirectUri);
20         idTokenResponse = IdTokenResponse.execute(tokenRequest);
21         return validateTokens(idTokenResponse, oidcConfig);
22     } catch (...Exception e) {
23         return Response...BAD REQUEST...;
24     }
25 }
```

Listing 5.3: Step 2 - Callback request using the Google library

Step 3 using the Google library is shown in Listing 5.4. Here all the required ID token checks [1] are implemented, using the various *verify* methods implemented in the IdToken wrapper class in the Google library. The ID token is parsed in line 4. The checks in lines 5-24 are similar, retrieving appropriate stored values, each returning error responses with the HTTP code 401 UNAUTHORIZED if the check fails. If none of the checks fail, the token response is stored in line 25, and a success response is returned with the token as payload in line 26.

```
1  // Step 3
2  public Response googleValidateTokens(... tokenResponse, ...oidcConfig) {
3      try {
4          IdToken idToken = tokenResponse.parseIdToken();
5          if(!oidcConfig.nonce.equals(idToken...getNonce())) {
6              return Response...UNAUTHORIZED...
7                      "Provided nonce did not match";
8          }
9          if(!idToken.verifySignature(publicKeyFromJwkSet())){
10             return Response.status(Response.Status.UNAUTHORIZED)
11                     "Jwt signature is not valid";
12         }
13         if(!idToken.verifyAudience(clientId)) {
```

[1] The required ID token checks are described in Chapter 2.4.1

```
            return Response...UNAUTHORIZED...
                   "Request not meant for this audience.";
        }
        if(!idToken.verifyTime(Instant.now(), TIME_SKEW_SECONDS)){
            return Response...UNAUTHORIZED...
                    "Token expired.";
        }
        if(!idToken.verifyIssuer(providerMetadata.get("issuer"))) {
            return Response...UNAUTHORIZED...
                   "The expected issuer did not match.";
        }
       ....createAndStoreCredential(tokenResponse, oidcConfig.appuuid);
        return Response.ok()
                .entity(tokenResponse)
    } catch (... | ... | ...Exception e) {
        return Response.status(Response.Status.BAD_REQUEST).build();
   }
}
```

Listing 5.4: Step 3 - Correct token validation using the Google library.

In Listing 5.5, the ID token verification is written using the Nimbus SDK. This code example is very different from the one for the Google library. In lines 5-9, the *IdTokenValidator* is instanciated with the required values, including encryption algorithms and other data from the Discovery document. The in lines 15-16, the store *nonce* parameter is retrieved and the ID token is obtained from the token response. Then by calling `idTokenValidator.validate`, the required checks are done by the validator, which in throws a *BadJOSEException* if any of the checks failed, and a *JOSEException* if an error happened during the validation. If the checks did not fail, a success response with the token request as payload is return in line 24. It is here appropriate to use the *IDTokenValidator* object provided by the SDK, since it does all the required checks if it receives the correct values from the developer.

```
1  // Step 3
2  public Response nimbusValidateTokens(...tokenResponse, ...oidcConfig) {
3      JWSAlgorithm metadataAlg = JWSAlgorithm.RS256;
4      try {
5          idTokenValidator = new IDTokenValidator(
6                  providerMetadata.getIssuer(),
7                  clientID,
8                  metadataAlg,
9                  providerMetadata.getJWKSetURI().toURL());
10                 // JWKsetUri gives the keys from the IdP
11     }  catch (MalformedURLException e) {
12         return Response...INTERNAL_SERVER_ERROR...
13                 "The provider metadata jwkSetUri is invalid";
14     }
15     Nonce expectedNonce = oidcConfig.nonce;
16     JWT idToken = tokenResponse.getOIDCTokens().getIDToken();
17     try {
18         idTokenValidator.validate(idToken, expectedNonce);
19     } catch (BadJOSEException e) {
20         return Response....UNAUTHORIZED)..."Invalid ID token";
21     } catch (JOSEException e) {
22         return Response...BAD_REQUEST..."Error validating ID token.";
23     }
24     return Response...200 OK...
```

```
            tokenResponse.toJSONObject();

}
```

Listing 5.5: Step 3 - Correct token validation using the Nimbus SDK.

5.2 Checklist for Authorization code flow implementation of RP

Rules for correct development can be worked out based on the OpenID Connect specification [60], the developer SDK guides by Nimbus and Google Api Client [37], and analysis of their javadocs and open-sourced code bases. These rules form a step-wise checklist rooted in the model in Section 5.1

5.2.1 Step 0: Establish trust with IdP

- Do Client Registration on an IdP console and registering an app client to generate keys and thereby establish trust.
- Receive client_id and client_secret from IdP console. This would be on an admin page on google's IdP or for example on Azure AD console, where you register a client in the UI. The client identifier and secret are generated by the IdP.
- Discovery process: Obtain IdP metadata (Listing 5.1).
  - Ensure TLS connection. Urls MUST use https (Lines 8-12).
  - Ensure only HTTP OK response from openid-configuration endpoint (Lines 13-15).
  - Proper parsing of the JSON object that is expected from the endpoint (Lines 16-20, and the `parseJson` method called in Line 21 [2]).
  - The saved JSON object is the *discovery* document, which is used to retrieve values that have integrity.

5.2.2 Step 1: Authorization code request (Listing 5.2

- Create state and nonce. These are proper opaque cryptographic random-generated strings or hashes. A sufficient entropy must be ensured. State is used to mitigate CSRF attacks, maintaining state between the authorization request and the callback. Nonce associates a client session with an ID token, and is used to mitigate replay attacks. The

[2]Example of JSON parsing is found here: https://github.com/Eliassoren/find-sec-bugs/blob/feature/evaluation-opensource/findsecbugs-samples-java/src/test/java/testcode/oidc/googleapiclient/OidcAuthFlowCompleteExampleGoogle.java

value is passed through from the authorization request to the ID token, and must be unmodified.

- Store state and nonce safely. Could probably store it in a cookie, HTTP session or for example retrieve them from a cache using a GUID associated with the request agent. Details about how to do this step is not in the scope of this work.
- Make an authentication request URI
  - Passing at least the parameters:
    - client_id
    - response_type
    - scope
    - redirect_uri
    - state
    - nonce
  - Using the Nimbus SDK
    - Build an instance of AuthenticationRequest with the required parameters. Use the Builder to add parameters like the login_hint (optional but recommended)
  - Using the Google SDK
    - Make an AuthorizationCodeFlow instance, adding client identifiers and urls for endpoints. Use AuthorizationCodeFlow.newAuthorizationUrl()
    - Authentication request made in a builder pattern in difference to strict type parameters in nimbus. A lot easier to forget state and nonce as you have to add them manually. In nimbus you explicitly have to pass null as state and nonce parameters to even run a request.
- Redirect the user agent with the authentication request URI to the authorization endpoint. Now the end user will log in on the IdP's side.

5.2.3 Step 2: Parse response and Token Request (Listing 5.3)

- Parse the response from IdP which comes as a POST request from the IdP.
- Handle any errors in the response. Break the flow if we have errors and return a HTTP 401 UNAUTHORIZED response.
- Verify that the stored state parameter matches the one in the IdP response. This is a simple equality test for the objects response.getState and stored.getState. If these are unequal, Http response(code=401) should be returned immediately.
- Retrieve authorization code from the success response. For example Call getCode() on the successResponse object.
- Send token request to token_endpoint with required parameters:
  - code,
  - client_id,

  - client_secret,
  - grant_type: “authorization_code”,
  - and redirect_uri

- Do proper error handling with token response: If you have error response the control flow must be broken, return a HTTP code 401 UNAUTHORIZED.
- With successful token response, parse the token response. Pass the ID Token on to step 3 for validation. The SDKs have implemented the parse function, use for example *TokenResponse.parse()*.

5.2.4 Step 3: Validate tokens (Listings 5.4 and 5.5)

In this step we have a more significant difference in use of the two SDKs for validating the ID token. There are a set of validation steps that are required [20, 60] to ensure the integrity of an ID token. Both the SDKs in this study have implemented an ID Token validation utility class, but they differ somewhat in their coverage of these validations requirements.

The parsed token response contains an ID token, an access token, and optionally a refresh token. Here we look at ID token validation.

- Using the Nimbus SDK:
  - The developer must retrieve and pass the following parameters to set up IdTokenValidator.
    * Issuer,
    * ClientID,
    * JWS Algorithm
    * JWK Set Uri (URL to the IdP’s JSON Web Key Set.)
  - Retrieve parsed idToken from the token response as JWT. The SDKs handle well-defined JOSE parsing for JSON Web Tokens.
  - Retrieve the stored nonce as the expected nonce for the anti-replay protection check.
  - Call the validate function in `IdTokenValidator` passing the passed ID token and the expected nonce.
  - The IdTokenValidator performs the following validations for the developer [16]:
    * Checks that ID token JWS algorithm matches the expected algorithm.
    * Checks the ID token signature or HMAC using the provided key material, from the client secret or JWK set URL in the discovery document.
    * Checks if the ID token *iss* and *aud* parameters match the expected IdP and client_id.
    * Checks that the ID token is within the specified validity window (between iat and exp time, given a 1 minute leeway to accommodate clock skew).
    * Check the nonce value in the request matches the expected one, if one is expected.

    * Therefore the developer does not have to perform any additional checks in order to follow the specification. However the above checks must be performed if you choose to implement them manually.

- Using the Google SDK:
  - Cryptographic signature validation and nonce validation must be done manually by the developer.
  - The `IdTokenVerifier` can be set up with parameters *issuer* and *client_id*.
  - The internal ItTokenVerifier performs the following checks [40]:
    * Check if the iss and aud parameters match the expected IdP and client_id
    * Checks if time is within acceptable validity window (exp and iat parameters) using idToken, with time skew leeway.
  - Even using the `IdTokenVerifier` the developer must either way verify:
    * That the JWS algoritm matches the expected retrieved from discovery document.
    * That the ID token signature is valid using the key from the discovery document.
    * That the nonce value matches the saved (expected) one.
  - Therefore for code clarity the developer should probably just validate everything that is recommended until the SDK implements all checks in a future release. The IdToken class [3] has implemented designated verify-methods for most of these.
    * That ID token JWS algorithm matches the expected algorithm.
    * The ID token signature or HMAC using the provided key material, from the client secret or JWK set URL in the discovery document.
    * If the ID token iss and audience aud parameters match the expected IdP and client_id.
    * That the ID token is within the specified validity window (between iat and exp time, given a 1 minute leeway to accommodate clock skew).
    * The nonce value matches the saved (expected) one.

    Optional: validate the Access Tokens After the ID token is validated, it may be used to obtain user info from the UserInfo endpoint.

5.2.5 Anti-patterns and bad practices

In addition to the pragmatic rules for good practice, there are also some obvious actions that can be considered unsafe practices or anti-patterns. The following usages of artifacts indicate smelly code or a bad practice:

- Usage of the *Resource Owner Password Grant*. In a later update of the OAuth standard [41], this grant is no longer considered acceptable.

[3] Google SDK: IdToken.java `https://github.com/googleapis/google-oauth-java-client/blob/master/google-oauth-client/src/main/java/com/google/api/client/auth/openidconnect/IdToken.java`

- Usage of a known limited ID Token “validator” provided by an SDK, like the IdToken-Verifier [40] of the Google library. This may trick the developer into thinking that all needed checks are done.

Furthermore, the checklist in Section 5.2 can be reversed to a set of vulnerabilities.

5.3 Vulnerabilities breaking rules in the model

The rules suggested in the checklist are all steps proposed to ensure security in the protocol, either from a MUST or SHOULD proposal in the protocol specification, or from security recommendations in other research. Breaking any one of the rules suggested here can be considered a risk or smell, if not necessarily a direct vulnerability.

However in this thesis any broken rule is considered a vulnerability, and the model is used to enforce a proposal of “proper practice”. It is stricter in its rules than the official specification, which has several important values that still stand as optional ones. Instead of focusing on the threats that are modeled from an attacker standpoint by previous works, like shown in Table 3.1, this thesis looks more closely on vulnerabilities in form of implementation errors.

Any implementation error cannot not necessarily be directly associated with one of the formally identified threats, but an error is still a broken implementation of how the protocol is intended.

Table 5.1 gives an initial overview of the vulnerabilities that developers can potentially introduce in their client code, given implementation errors. This list is not exhaustive, and lot more vulnerabilities related to implementation details may be inferred using the model.

Table 5.1: Potential vulnerabilities as various errors that can occur by breaking the rules in the model.

Step in the developer model	Potential vulnerabilities
0. Discovery	Not ensured HTTPS connection, and fail to break flow if not https. Error handling if response not 200 OK Improper JSON parsing.
1. Authentication Request	Not obtaining IdP Metadata with discovery protocol Not adding state and nonce to request or not storing state and nonce Not obtaining IdP Metadata with discovery protocol
2. Callback	Improper error handling Missing State verification Not obtaining IdP Metadata with discovery protocol
3. Token parsing	Missing required ID Token checks Usage of known incomplete SDK token validator Not obtaining IdP Metadata with discovery protocol Incorrect control flow in checks with an incorrect response to a failed condition

Chapter 6

RQ2 results: Design and implementation

The following chapter relates to RQ2: How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?. This chapter goes through the design and implementation of simple static analysis techniques for enforcing the security principles in the model for OpenID Connect (See Chapter 5.1). The idea is that analyses of three layers can cover a lot of the security-critical protocol steps the developer has to implement. Analyses are implemented as FindSecBugs detectors [1].

The first layer is using the `OpcodeStackDetector` analysis from FindBugs (See Chapter 2.7), here named the *Immediate Code Smell Detection* analysis, which only looks at a single instruction in the JVM bytecode. The second layer is the *Co-existing Invocation Enforcement* analysis, which reasons about each method in a class, as well as inter-procedural approximation. Lastly, the third layer is the *Static Control Flow Check* analysis, which analyzes the control-flow graph.

The focus here is vulnerabilities in code calling OpenID Connect SDKs, meaning bugs that developers may introduce when they write code that interfaces with these SDKs. The analyses are not concerned with looking for vulnerabilities in the SDKs themselves.

6.1 Definition of analysis terms

Simple terms are established to define the scope of the analyses in this context.

Definition 6.1.1. Peephole: In this thesis, a peephole is an instruction in the program's bytecode, denoted as <x>, a CFG edge type or a combination of instructions in a java method. A peephole can be considered a simple property that is used to infer a property in a more complex flow.

Definition 6.1.2. Peephole pattern <pat>: An expected single or combined collection of peepholes in the bytecode that indicate an action in the protocol is executed. For exam-

[1]The concept of detectors is described in Chapter 2.7.

ple, simply the invocation of the class `TokenResponse` would imply that we are in a method implementing step 3 of the code flow in Listing 5.4.

Identifiers used by the analyses are defined in Table 6.1.

Table 6.1: Definitions for identifiers that are used in the checks of the peephole analyses.

Peephole	**Description**
<inv>	Invocation <inv> is defined as a bytecode *invoke* instruction in which a certain class or type is instantiated.
<cmp>	Comparison <cmp> is defined as an if-instruction like the `ifne` bytecode instruction.
<ret>	Return <ret> is defined as the act of returning a certain HTTP Response code or throwing an exception.
<ver> *b*	Verification <ver> of a value *b* either happens with an <inv> with *b*.equals(), or as *b* passed to another method in which an <inv> with *b*.equals is called.
<comb>	Combination <comb> is the coexistence of a set of peepholes in the bytecode.
<pat>	Peephole Pattern <pat> is in this context the appearance of one of or a combination of the attributes <cmp>, <inv>, <ver> or <ret>.
<pair>	A strict pair of patterns where we expect pattern *b* to be found if we have found pattern *a*.

6.2 Detector types

Some of the pattern-matching techniques defined in FindBugs and used in FindSecBugs (See Chapter 2.7) like *OpcodeStackDetector* and *CFG-based detector* were adopted in the analyses for the potential vulnerabilities. The detector types defined in this thesis are divided in three types, with increasing sophistication level: 1), Immediate Code Smell Detection, 2) Co-existing Invocation Enforcement, and 3) Static Control Flow Check.

6.2.1 Immediate Code Smell Detection

The immediate code smell detectors utilize the same technique as the *OpcodeStackDetector* defined in FindBugs, and is the simplest of the three used in this study. This technique is not new, but fits well together as the simplest component together with the two other analyses proposed in this thesis. The detector is based on that the prevalence of a given peephole pattern indicates smelly code. An assumption here is that it does not necessarily find a true bug, we just flag some smelly code to raise warning and inform the developer. This could for instance just be the usage of a data type which is associated with a disallowed pattern in the protocol.

Both its strength and its weakness lies in this simplicity. The way it is used in other parts of FindSecBugs, we have a black-list of known functions that must never be used. For instance just using the `Math.random()` function is something that typically must not be seen in production code. It is therefore enough to just flag this value as a code smell, and make sure that the developer is informed. The basic algorithm for the detector is quite simple. Define a set of peephole patterns <pat>, which usually would be <pat> = {<pat1>: <inv> type A, <pat2>: <inv> type B}. Then the typical detector is implemented like shown in Algorithm 6.1:

```
input: Code File
output: Vulnerability Reports
begin
    scan opcodeStack in Code File
    foreach opcode in opcodeStack
        if opcode in <pat>
            report vulnerability <pat>
        end
    end
end
```

Algorithm 6.1: Basic strategy for Immediate Code Smell Detection

Limitations of the analysis

This analysis is limited to very simple facts about a single instruction, and cannot infer more complex relations between data items. It can however flag a data type that is associated with a code smell.

6.2.2 Co-existing Invocation Enforcement

Ensure that the existence of a certain peephole pattern <pat_b> in a method, happens if the peephole pattern <pat_a> is there. This can be described as <$pair_1$> = <pat_a> ∧ <pat_b>. This detector linearly scans through the bytecode instructions in each method, and effectively ignores control flow and data flow. For instance <pat_a> could be that we receive some data from a http request, while <pat_b> is a certain verification step that is needed in the protocol. Therefore we would only expect to find <pat_b> if <pat_a> has already been satisfied. This means that if <pat_a> has been satisfied, <pat_b> MUST also have been found while the instructions were analyzed. This is illustrated in Algorithm 6.2:

```
input: Code File
output: Vulnerability Reports
begin
    foreach method in Code File
        foreach instruction in method
            if instruction matches <$pat_a$>
                found_pat_a = true
            end
            else if instruction matches <$pat_b$>
                found_pat_b = true
            end
        end
        if found_pat_a and not found_pat_b
            report vulnerability Missing (<$pair_1$>)
        end
    end
end
```

Algorithm 6.2: Basic strategy for *Co-existing Invocation Enforcement* detectors.

While the analysis is inherently intra-procedural, it as has a small inter-procedural component. It can note methods of suspicion for an additional scan with some more information in the end of the "main" analysis. These extra pick-up-leaves analyses are central in handling one of the most highly expected patterns in validation code - that a check is delegated to another method.

Secure code enforced

Some simple examples can illustrate some of the abilities and boundaries of this analysis. For example the analysis could expect to see that if `b()` has been called, somewhere `c()` must follow. The code below would then be passed as safe, thereby a true negative:

```
void a() {
    b();
    ... other code
    c()
}
```

It will also let code like below pass as safe, where the call to `c()` is delegated to another method `d()`:

```
boolean d() {
    c()
}
void a(var) {
    b();
    ... other code
    d()
}
```

Vulnerable code which raises warning

After `b()` has been called, somewhere `c()` must follow. However something else is there, but `c()` is missing. The snippet below would then raise a warning:

```
void a() {
    b();
    ... other code
    e();
    return f;
}
```

The call to `c()` may be delegated to another method `d()`. However `d()` does not have any call to `c()` either, even if its name and context would suggest so. This is therefore vulnerable.

```
boolean d() {
    e(); // c() is missing!
    return f;
}
void a() {
    b();
    ... other code
   return d();
}
```

Here the inter-procedural component will come in. In this case the subsequent enforcement follow this strategy to detect that we have a broken rule:

1. Scan through the list of methods in a class.
2. Note that `a()` has a called on `b()`, which means that somewhere in this area `c()` must be found.
3. Scan linearly through the method. `c()` was not found, but `d()` may have a call to `c()`, indicated by its parameters and name. Save a pair of `a()` and `d()` for later inspection.
4. Scan further through the rest of the methods and finish the list. If any method contains a call to `c()`, save it in a list of *approved* methods.
5. Scan through the methods which have a suspected call to another, which may contain a call to `c()`. Then finish the list.
6. Look through the methods saved for later inspection. `d()` is expected to have a call to `c()`. Check if `d()` is in the list of approved methods. If not, raise warning on `a()`.

This component is introduced because of a much-seen code pattern where a check is not done in-line, but delegated to a pure *verify* method.

Vulnerable code which the analysis cannot pick up

The only thing the Co-existing Invocation Enforcement really does, is looking at the existence of certain method calls in the code. It however makes no assumption on whether they are done right in terms of control flow. The following code would be vulnerable, but come as a false negative using the Co-existing Invocation Enforcement:

```
boolean isValid(data){
    if(data.c() != safe) {
        return true;
    }
    return false;
}
```

Here what the Co-existing Invocation Enforcement would look for is the comparison of `data.c()` and another value. However the developer of this code has made a blatant mistake and reversed the if conditional, so that in any case where `data.c()` is not safe, is says that it is safe. Such cases are therefore of too a subtle nature for this analysis. To deal with easy control-flow mistakes, the *Static Control Flow Check* is appropriate to use (see Section 6.2.3)
.

Limitations of the analysis

This detection strategy is simple, but is limited to detecting the appearance or absence of usages of a certain data attribute, i.e. the state parameter or the ID token. It has a simple inter-procedural component that covers a simple, and easy-to expect case of delegating a check downwards to another method. However it does not reason completely about inter-procedural artifacts, and if the methods are structured in a special way in the code, it may miss vulnerabilities or give false positives.

Another limitation is that it is unable to tell whether the check is carried out with a proper control flow (the check may be useless if it does not enforce what happens *after* the check). However this strategy is cheaper. The absence of the checks this analysis looks for makes a heavier control flow analysis unnecessary, since there then would be no control flow to check. Only when the checks are present, control flow analysis can be employed to further verify the solidity of the code.

6.2.3 Static Control Flow Check

The goal of the *Static Control Flow Check* is to ensure that every basic block ending with a conditional check of a verification call leads to a block returning with a correct error response. This is relatively naive approach to control flow and data flow, only specifying peephole patterns that may be satisfied by certain basic blocks in the control flow graph. It can be used in cases of the model where developers carefully have to ensure a proper control flow, and is highly specialized and constrained towards what is a valid pattern. This is made in the spirit of FindBugs [44], with the notion that developers make "dumb" mistakes like reversing an if conditional or continuing running the code after a catch block where an error case should be managed.

The initial assumption in a simple conceptual CFG is that you have a series of checks that divide control flow. A typical pattern discovered in the validation steps of the OIDC protocol for this model, are quite linearly placed if-conditionals that ends the program right there with an error code or continues the flow if the check passed. This technique is applicable for relatively linear control-flow graphs which follow one consistent green path and otherwise break off early.

Such a graph is illustrated in Figure 6.1. Here the basic control flow of the code example for token validation (Listing 6.11) is shown in a simplified manner. The main point of the code in Listing 5.4 is that you have a series of if-else checks of values in the ID token. These if-else checks can be modeled as a simple binary control-flow graph which either ends in a leaf node or goes further down the thee.

In Figure 6.1, the green boxes represent a successful check. If one of the if-checks correctly verified the value, it will go to the next if-check. However if the checked value is invalid, the control flow is broken. Then we end up in a red box in the modeled graph, thus ending the control flow path and stopping the flow in the program.

During the analysis the intuition is to look for a blue block, which should be a negative comparison like `!a.equals(b)`. Such a block is followed by expected outgoing edges, and the fall-through edge leads into the if-block braces. If we are inside the if-block it meant one of the verification steps has failed, and the ID Token is invalid. We therefore expect a return statement which takes us out of the method. This return statement is also expected to give an appropriate HTTP error code.

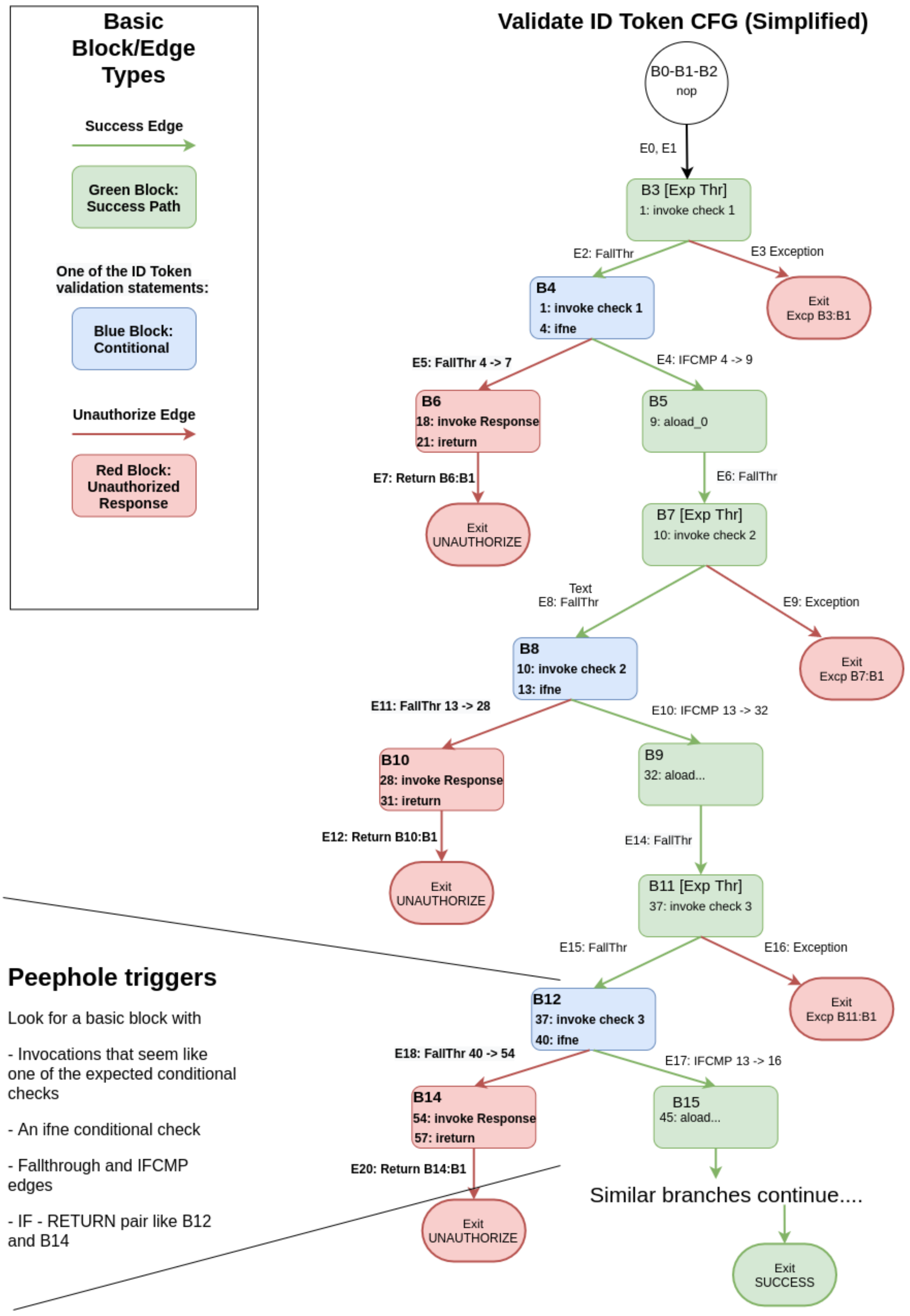


Figure 6.1: A simplified control flow graph of token verification.

Given the above, the intuitive peephole check is a simple look at a basic block, and the neighboring block following one of the outgoing edges. However in a code base with several invocations, the analysis is not quite so simple as to look at a single edge between the if-statement and the code in its following curly braces. Rather, if we return the object: `Response.status(CONSTANT...)`, a chain of invocations happen as we perform several compact method calls. Therefore the reality is a more complex graph like shown in Figure 6.2. Here we actually have to traverse a series of "leaf" nodes to get to the actual leaf node.

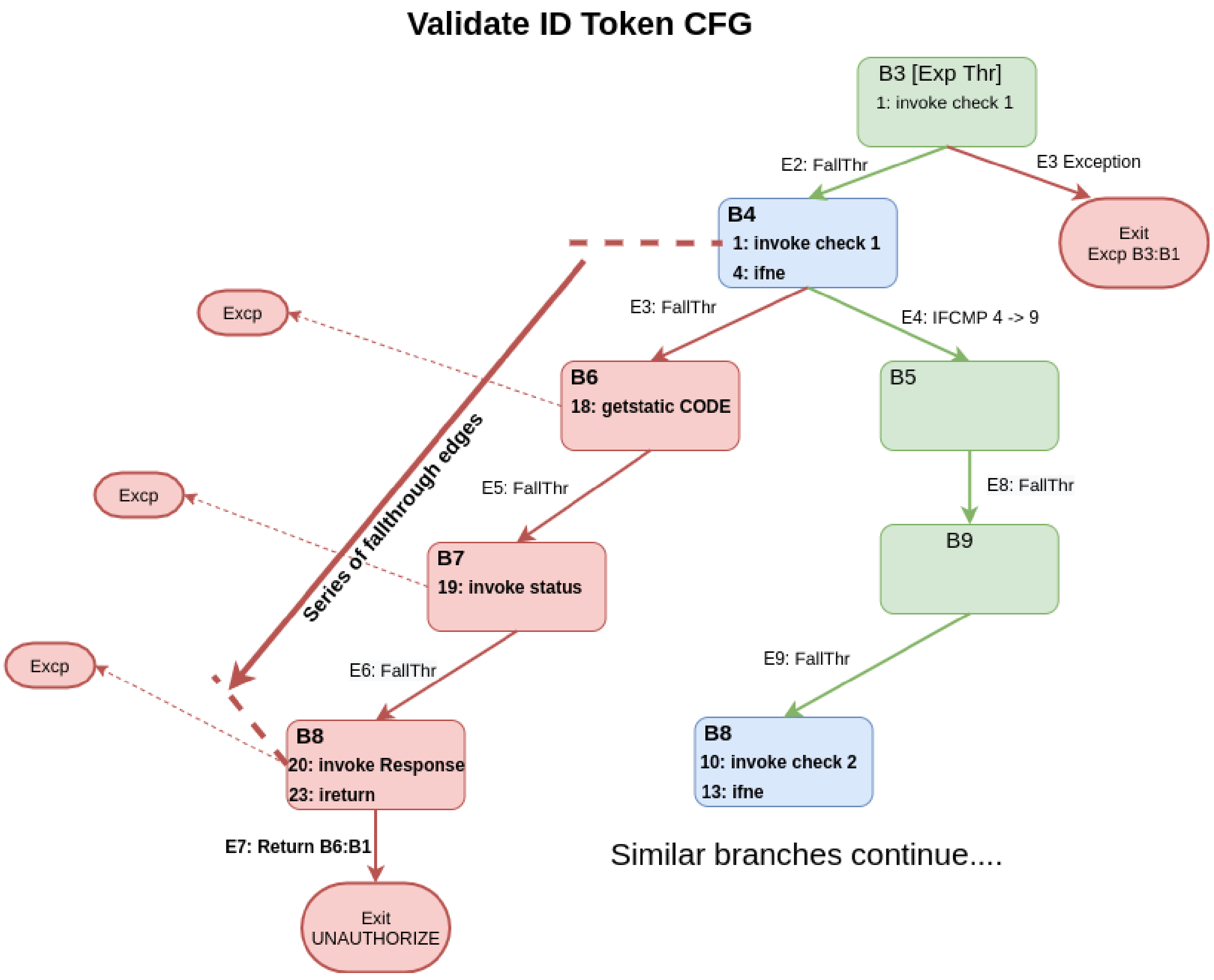


Figure 6.2: A closer look at the control flow graph for token verification.

This means that even a minimized peephole analysis needs to traverse the CFG to some extent. Luckily, the control flow graph in such return statements as described above are still relatively simple. Each of the basic blocks have usually two outgoing edges, either an exception edge if the invocation failed, or simply fall-through to the next invocation. This series of basic blocks could therefore almost be considered parts of the edge between our two main trigger points, namely the blue *conditional* block, and the final red *return block*. However, simple hints may come from the instructions also inside these "fall-through" blocks. These hints are tracked in a simple data object that is checked in the end of the analysis. Even while traversing the CFG and performing some operations that approximate data-flow, the triggers of the analysis are simply a look-and-match. For certain expected instructions inside each basic block, and a combination pattern of these basic blocks in a single method.

The detection strategy follows, as shown simply in Algorithm 6.3: Iterate the basic blocks in the CFG. If a block contains one of our expected verification method calls, and it is an if/else conditional block, we have triggered the analysis. Traverse linearly through the *fall-through* edges following the conditional check. Pick up additional instructions in the traversed basic blocks. The sum of the peepholes in the series of basic blocks determine the peephole. If we do not find a following set of blocks that satisfies the expected return patterns, but end up in a new check or a different return state, we have a control flow bug.

```
input: Code File
output: Vulnerability Reports
begin
    foreach basicBlock in CFG
        if basicBlock instructions match <$pat_a$>
            traverse neighboring blocks looking for <$pat_b$>
            if not found instructions with <$pat_b$> in neighboring blocks
               report vulnerability
            end
        end
    end
end
```

Algorithm 6.3: Detection strategy for Static Control Flow Check in token validation bugs.

Limitations of the analysis

The Static Control Flow Check analysis has some of the same limitations as the other analyses. To avoid false positives, a large number of patterns are needed. It also suffers from lacking ability to track data, and will therefore be limited in how subtle errors it can find. A potential fourth detector type is to use data-flow analysis for another layer of sophistication. Such detectors exist in FindBugs and FindSecBugs for other vulnerabilities, and can potentially be designed for appropriate cases in OpenID Connect. A detector such as the ResourceTrackingDetector (See Chapter 2.7) may have some uses for certain cases in OpenID Connect where it is crucial to have a comprehensive view of the data flow. That is however beyond the scope of this study and is an interesting avenue for further work.

6.3 Detector types for different vulnerabilities

To make the production of new detectors light-weight, easy to understand and fast to implement, the minimum sophistication level needed to detect a vulnerability is selected. If, for instance, the vulnerability is usage of an anti-pattern that can be detected by a simple invocation in the bytecode, there is no need to compute a complex model of the program for the check. In several cases a fair assumption is that if the developer has included all the required checks, he has a good chance of doing it right. Still mistakes are easy to make, therefore in some other cases, one might have to reason about the control-flow graph to be more certain that a check is secure.

The model in Chapter 5.1 proposes a developer-oriented way to think about how OpenID Connect is to be implemented securely. Based on the model, a set of potential vulnerabilities due to implementation bugs were inferred in Table 5.1. The three detector types are appropriate for different kinds of bugs in OIDC code, and similar algorithms may work for similar issues related to different data items.

Table 6.2 shows a suggestion of which vulnerabilities can be detected by which kind of analysis. Many of these vulnerabilities will occur for similar reasons, making similar detectors appropriate for covering the whole authorization code flow. FindSecBugs detectors were implemented for four of these inferred bugs. These implemented detectors are explained in Section 6.2.

Table 6.2: Suggested analyses to be implemented as detector for various errors that can occur by breaking the rules in the model. The detectors that are implemented in this study are highlighted with **bold**

OIDC step	Implementation errors	Detector type
0. Discovery	Not ensured HTTPS connection, and fail to break flow if not https.	Static control flow peephole.
	Error handling if response not 200 OK	Static control flow peephole
	Improper JSON parsing	Immediate code smell
1. Authentication Request	Not obtaining IdP Metadata with discovery protocol	Subsequent invocation
	Not adding state and nonce to request or not storing state and nonce	Subsequent invocation
	Not obtaining IdP Metadata with discovery protocol	Subsequent invocation
2. Callback	Improper error handling	Static control flow peephole.
	Missing State verification	**Subsequent invocation** (Implemented Section 6.4.2)
	Not obtaining IdP Metadata with discovery protocol	Subsequent invocation
	Using insecure authorization grant	**Immediate code smell** (Implemented Section 6.4.1)
3. Token parsing	Missing required ID Token checks	**Subsequent invocation** (Implemented Section 6.4.3)
	Usage of known incomplete SDK token validator	Immediate code smell
	Not obtaining IdP Metadata with discovery protocol	Subsequent invocation
	Incorrect control flow in ID token verification checks with an incorrect response to a failed condition	**Static control flow peephole** (Implemented Section 6.4.4).

6.4 Implemented detectors

The following detectors that have been implemented for the demonstration of the three analyses:

- **Immediate Code Smell Detection:** Insecure authorization grant detector (Auth. Gr.), which is explained in Section 6.4.1.
- **Co-existing Invocation Enforcement:** Improper state verification detector (State ver.), which is explained in Section 6.4.2.
- **Co-existing Invocation Enforcement:** Improper ID token verification detector (Token ver.), which elaborated in Section 6.4.3.
- **Static Control Flow Check:** Token CFG (Token CFG), which is explained in Section 6.4.4.

In FindBugs, each detector reports a set of *bug patterns*, or in this context also referred to as vulnerability patterns. These patterns are created based on the protocol model, and the potential vulnerabilities in Table 5.1. A bug pattern may relate directly to these potential vulnerabilities, but do in cases offer various level of detail. This is a mapping between the theoretical model, and practical ways to report and inform developers of their vulnerabilities. Table 6.3 gives an overview of the main vulnerability patterns for the detectors. Specific vulnerability patterns are presented for each implemented detector respectively, in Tables 6.4, 6.5, 6.6 and 6.7.

Table 6.3: Overview of the main *vulnerability patterns* developed for the implemented detectors. These are explained in more detail under each detector in Sections 6.4.1 to 6.4.4

Detector	Main patterns	Description
Auth. Gr.	USING_PASSWORD _GRANT_OAUTH	Usage of the *Resource Owner Password Grant*.
State ver.	MISSING_VERIFY_OIDC_STATE	Not having a check of *state* value in callback.
Token ver.	MISSING_VERIFY _ID_TOKEN	All five token checks are missing.
	INCOMPLETE_ID _TOKEN_VERIFICATION	The developer has implemented verification but misses some checks.
	USING_INCOMPLETE _ID_TOKEN_VALIDATOR	Known incomplete SDK-imlemented *ID token validators* are used.
Token CFG	IMPROPER_TOKEN_VERIFY _CONTROL_FLOW	The control flow response to a security check is not as expected.
	REVERSED_IF_EQUALS _ID_TOKEN_VERIFY	Possible reversed conditional in a security if-check.

6.4.1 Insecure authorization grant detector

Like described in Section 6.2, what this kind of detectors seeks to find is not necessarily a true vulnerability, but rather a smell in the code, which means the developer should be informed in case they unknowingly use types associated with anti-patterns. The Resource Owner Password Grant is disallowed in the newer version of OAuth [41].

Secure code example

Listing 6.4 shows a secure code example of the authorization grant. The authorization code grant is considered secure when implementing OpenID connect.

```
AuthorizationGrant codeGrant = new AuthorizationCodeGrant(
                                         authorizationCode,
                                         callbackURI);
TokenRequest tokenReq = new TokenRequest(
                        providerMetadata.getTokenEndpointURI(),
                        clientSecretBasic,
                        codeGrant,
                        scopes
                        ...)
```

Listing 6.4: Correct usage of authorization grant, like using the authorization code grant.

Vulnerable code example

An example of a vulnerable grant flow is shown in Listing 6.5, in which the *Resource Owner Password Grant* is used in line 2. The detector will raise a warning if encountered with code where objects like the *ResourceOwnerPasswordCredentialsGrant* are used, because these denote usage of the unsafe grant type.

```
1   AuthorizationGrant
2            passwordGrant = new ResourceOwnerPasswordCredentialsGrant(
3                                               username,
4                                               password);
5   TokenRequest tokenReq =   new TokenRequest(
6                               tokenEndpoint,
7                               clientSecretBasic,
8                               passwordGrant,
9                               scopes,
10                             ...)
```

Listing 6.5: Usage of the Resource Owner Password Grant.

Detection strategy

This detector simply uses the behavior of the OpCodeStackdetector (See Chapter 2.7). It looks for a bytecode instruction which matches its blacklist, which consist of types that denote usage of the bad code grant. When a match comes it raises a warning. Otherwise it will ignore everything, so it is not very prone to false positives.

Vulnerability Patterns

The Insecure authorization grant detector currently looks at one vulnerability class. This class is reported if it notices the usage of a bad authorization grant:

Table 6.4: Vulnerability patterns in the *Insecure authorization grant detector*.

Pattern	Abbreviation	Description
USING_PASSWORD _GRANT_OAUTH	SECISAUTH	Usage of the *ResourceOwnerPasswordCredentialsGrant* type in the Nimbus SDK, or the *PasswordTokenRequest* in the Google library.

6.4.2 Improper state verification detector

When doing an authorization request in OpenID Connect, the client always has to verify that the state in the response matches the state you sent with the request. If such a check is absent, the code has a vulnerability, and can be exposed to CSRF attacks.

Secure code example

In step 1, an authentication request has added the *state* parameter to the authentication request (See Listing 5.2). This value must be checked in step 2, which is receives the callback request. A secure code example is shown in Listing 5.3.

Vulnerable code example

The error of state checking is quite simple. In the code in Listing 6.6, the state is not checked even if we are in the callback context. Here the code proceeds to use the authorization code in line 10.

```
1  public Response callback(HttpServletRequest req) {
2     try {
3          UUID uuid = UUID.fromString(...req.get(uuid));
4          OidcConfig oidcConfig = (OidcConfig)cache.get(uuid);
5          ...ResponseUrl responseUrl = new ...Url(req.getRequestURI());
6          String error = responseUrl.getError();
7          if(error != null) {
8              return Response...UNAUTHORIZED...;
9          }
10         // Missing check state!
11         String authorizationCode = responseUrl.getCode();
12         .... token response
13 }
```

Listing 6.6: Vulnerable state usage. Forgets to check state!

Vulnerability Patterns

There are two vulnerability patterns defined in this detector, shown in Table 6.5.

Table 6.5: Vulnerability patterns in the *Improper state verification detector*. In addition to the blatant missing verification of the state parameter, passing the value somewhere unvalidated is flagged as a lower-level warning.

Pattern	Abbreviation	Description
MISSING_VERIFY_OIDC_STATE	SECVMOS	Not having a check of *state* value after an AuthenticationResponse is parsed.
EXTERNAL_CALL_POSSIBLY _MISSING_VERIFY_OIDC_STATE	SECVMOSEXT	The state value is not verified in-line, but is passed to some method which is unreachable by the detector. This is possibly not a verification method.

Detection strategy

To detect bugs related to improper validation of the state parameter, we have the following strategy (which is further explained in Section 6.2): Scan through each method in the code file, identify a method call that is an authentication response. For each such method call, identify an action that compares an existing *state* string to the *state* parameter retrieved from the authentication response. The absence of such a comparison means we have a potential vulnerability, and MISSING_VERIFY_OIDC_STATE. Additionally, identify if the State object is passed to another method. If we cannot find verification in the called method, report MISSING_VERIFY_OIDC_STATE. If the called method is not in this Java class, and no checks were found elsewhere, report EXTERNAL_CALL_POSSIBLY_MISSING_VERIFY_OIDC_STATE

Severity

This bug may lead to Replay Attacks (See Table 3.1). An attacker may impersonate a protocol entity by obtaining a credential value. However validation of the state parameter helps mitigate this kind of attack, as several separate data artifacts contribute to the integrity of a request.

6.4.3 Improper ID token verification detector

The Improper ID token verification detector detector looks for improper verification of ID Tokens in step 3 of the model (Chapter 5.1). To enforce rules based on the model and checklist in Chapter 5, it expects the five values in the ID token to be verified. These may either be verified by checking the values from the ID token directly, using an SDK-developed ID token object implementation checking these values, or an *ID Token verifier* utility class offered by the SDK.

Secure code examples

There are generally two correct ways of implementing the ID Token verification in the development model. This step differs from SDK to SDK. In google you have two options. Either use the token validator and check the last two (signature and nonce) yourself, or call the `validate`-methods implemented on the `IDToken` wrapper class around JWT.

Listing 6.7 shows how the token request is constructed in the callback method, after receiving and verifying the callback request from the IdP. After running receiving the token response from `IdTokenResponse.execute(tokenRequest)` in line 10, the token response is passed to the verification method in line 11, and the process moves to step 3.

```
1  public Response callback(HttpServletRequest callbackRequest) {
2          try {
3              ...
4              // .. state validation and error check
5              String authorizationCode = responseUrl.getCode();
6              TokenRequest tokenRequest = authorizationCodeFlow.
   newTokenRequest(authorizationCode)
7                      .setTokenServerUrl(new GenericUrl(
   authorizationCodeFlow.getTokenServerEncodedUrl()))
8                      .setClientAuthentication(authorizationCodeFlow.
   getClientAuthentication())
9                      .setRedirectUri(redirectUri);
10             IdTokenResponse idTokenResponse = IdTokenResponse.execute(
   tokenRequest);
11             return validateTokens(idTokenResponse, oidcConfig);
12         } catch (Exception e) {
13            return Response.status(Response.Status.UNAUTHORIZED).build();
14         }
15
16     }
```

Listing 6.7: Step 3 - Token request with call to step 3, validateTokens.

In Listing 6.8, a correct example using the IdTokenVerifier is shown. If the developer manages to do the other required checks in addition to the initially incomplete validator, this is predicted as secure code.

```
// Step 3
public Response validateTokens(... tokenResponse, ...oidcConfig) {
    try {
        IdToken idToken = tokenResponse.parseIdToken(); // Parse
        IdTokenVerifier verifier = new IdTokenVerifier.Builder()
                                       .setAudience(clientId))
                                       .setIssuer(providerMetadata.get("
    iss"))
                                       .setAcceptableTimeScewSeconds(
    TIME_SKEW_SECONDS)
                                       .build();
        IdToken idToken = IdToken.parse(new GsonFactory(), tokenString);
        if(!oidcConfig.nonce.equals(idToken...getNonce())) {
            return Response...UNAUTHORIZED...
                    "Provided nonce did not match";
        }
        if(!idToken.verifySignature(publicKeyFromJwkSet())){
            return Response...UNAUTHORIZED...
```

```
                        "Jwt signature is not valid";
        }
        if(!verifier.verify(idToken)) {
            Response...UNAUTHORIZED...
        }
    } catch (Exception e) {
      return Response...UNAUTHORIZED...
    }
}
```

Listing 6.8: Step 3 - Correct token validation using the Google library. Full example in Listing 5.4. A corresponding example using the Nimbus SDK is shown in Listing 5.5.

Vulnerable code examples

The easiest example which will yield a warning is a received and parsed ID token response, without any following verification (all five checks missing) is shown in Listing 6.9. Verification of the ID token is expected between line 6 and line 9. This example is a vulnerability of type MISSING_VERIFY_ID_TOKEN.

```
 1 public Response callback(HttpServletRequest callbackRequest) {
 2         try {
 3             // After verified state and parse auth code..
 4             TokenRequest tokenRequest = ...
 5             IdTokenResponse idTokenResponse = IdTokenResponse.execute(
      tokenRequest);
 6             IdToken idToken = idTokenResponse.parseIdToken();
 7             // BUG: missing verification
 8             // userinfo request with ID token...
 9             return Response.ok()
10                     .entity(idTokenResponse)
11                     .build();
12             }
13         } catch (Exception e) {
14             // Error handling
15         }
16         return Response...UNAUTHORIZED...
17     }
```

Listing 6.9: Step 3 - Missing token validation using the Google library.

Listing 6.10 shows an example which is of type INCOMPLETE_ID_TOKEN_VERIFICATION, as it is missing checks of the nonce and signature (How these are checked is shown in Listing 6.8). The vulnerability pattern USING_INCOMPLETE_ID_TOKEN_VALIDATOR will also be raised, since the `IdTokenVerifier` is used without additional checks.

```

public Response callbackTokenVerifier(HttpServletRequest callbackRequest
    ) {
try {
    // After verified state and parse auth. code..
    TokenRequest tokenRequest = ...
    IdTokenResponse idTokenResponse = IdTokenResponse.execute(
   tokenRequest);
    IdTokenVerifier
```

```
        idTokenVerifier = new IdTokenVerifier
                                .Builder()
                                .setAudience(clientId))
                                .setIssuer(providerMetadata.get("iss"))
                                .setAcceptableTimeScewSeconds(
    TIME_SKEW_SECONDS)
                                .build();
    IdToken idToken = idTokenResponse.parseIdToken();
    if(idTokenVerifier.verify(idToken)) {
        // BUG: verifier is missing nonce and JWT signature check
        ...createAndStoreCredential(idTokenResponse, ...
        return Response.ok()
                .entity(idTokenResponse)
                .build();
    }
} catch (Exception e) {
    // Error handling
    return Response.status(Response.Status.UNAUTHORIZED).build();

}
}
```

Listing 6.10: Step 3 - Incomplete token validation using the Google library ID token verifier. This is another variation where this method is called from "callback".

Vulnerability Patterns

There are seven vulnerability patterns in this detector, shown in Table 6.6.

Table 6.6: Vulnerability Patterns in the *Improper ID token verification detector.* Two of them are collector classes for the token verification parameters. Seven individual vulnerability patterns are checked for in the detector.

Pattern	Abbreviation	Description
MISSING_VERIFY _ID_TOKEN	SECMVIDT	Collector class if all five token checks are missing.
INCOMPLETE_ID _TOKEN_VERIFICATION	SECIIDTV	Collector class to notify that not all checks are there.
MISSING_VERIFY_NONCE	SECMVNONCE	Missing check of *nonce* parameter.
MISSING_VERIFY_TOKEN_ISS	SECMVTISSU	Missing check of *iss* parameter.
MISSING_VERIFY_TOKEN_AUD	SECMVAUD	Missing check of *aud* parameter.
MISSING_VERIFY_TOKEN_SIGN	SECMVTSIGN	Missing check of JWT *signature.*
MISSING_VERIFY_TOKEN_EXP	SECMVTEXP	Missing check of *iss* parameter.
EXTERNAL_CALL_POSSIBLY _MISSING_VERIFY_ID_TOKEN	SECMVIDTEXT	The ID token is not verified in-line, but is passed to some method which is unreachable by the detector. This is possibly not a verification method.
USING_INCOMPLETE _ID_TOKEN_VALIDATOR	SECUIDTV	Integrated immediate code smell detector. Warns of known incomplete SDK-imlemented ID token verifiers. If checks are added elsewhere in the class, this will be suppressed.

Detection strategy

This detector also uses the Co-existing Invocation Enforcement analysis to detect vulnerabilities. The detection strategy uses the following process visiting each method in a Java class:

1. As we scan through the methods we collect relevant methods in a set of data structures:
    - The methods that have all required token checks.
    - The methods that require later inspection to search for all token checks.
    - A hash map of method pairs, the *caller* and the *called.* This for an inter-procedural approximation.
2. Look for invocations patterns that indicate that an ID token is retrieved, for example *TokenResponse.getIdToken().*
3. Look for other patterns that indicate that validation is happening. We have the following options:

   - An “IdTokenVerifier” is instantiated. If this verifier is in the black-list of incomplete verifiers, add the method for later inspection. If the missing checks are implemented aside from this class, this method is considered safe. If the SDK verifier is in the white-list of safe verifiers, the method is added to list 1.
   - One of the required checks is instantiated. Add the method to list 2.
   - The method passes the ID token to another method. This method and the method calls are added to the hash map of pairs for later inspection.

4. If the ID token was retrieved, but this method is not added to a list for later inspection, raise warning of MISSING_VERIFY_ID_TOKEN. No sign of validation was found.

5. Look through all the methods in list 2, searching for the five required checks to be present. If any one of the required checks is absent, raise a warning of INCOMPLETE_ID_TOKEN_VERIFICATION.

6. Look through the hash map of pairs. If the called method is not in list 1, raise a warning:

   - If the called method indicates that it attempts verification, report INCOMPLETE_ID_TOKEN_VERIFICATION.
   - If the called method has no such indication, report MISSING_VERIFY_ID_TOKEN.
   - If the called method is not in this Java class, and does not have a name that indicates that it does verification, report EXTERNAL_CALL_POSSIBLY_MISSING_VERIFY_ID_TOKEN. We have no other signs of validation, and the token is passed somewhere.

Severity

This bug may lead to Replay Attacks (See Table 3.1). An attacker may impersonate an adversary by obtaining a credential value. However validation of the state parameter helps mitigate this kind of attack, as several separate data artifacts contribute to the integrity of a request.

Limitations of the detection strategy

This detection strategy is simple, but is limited to detecting the appearance or absence of usages of a certain data attribute, namely the State parameter. Is is not able to tell whether the check is carried out in a proper control flow. However this strategy is cheaper, and the existence of this vulnerability makes a heavier control flow analysis unnecessary. Only when this bug does not exist, control flow and potentially data flow analysis can be employed to further verify the solidity of the code.

6.4.4 Control flow ID token verification detector

During token validation there are many things that may go wrong. The developer has to ensure that the correct data is validated using the correct control flow, with appropriate responses to specific checks. Generally the code must be really simple, with one single green path leading to a validated token, thus ending in a HTTP OK state and completed step. Any deviating path must end with a broken control flow and often the return of a 401 UNAUTHORIZED HTTP error code.

Secure code example

In code doing token validation, each check must be followed by a correct response. If a value does no match, a HTTP response code 401 UNAUTHORIZED must be returned, like shown in Listing 6.11. Here the checks are implemented as a series of similar if-checks. All the checks use a *negative* comparison, meaning that the value being true means that the values do not match. In line 4 for example, the program flow is broken and a response code is returned if the saved *nonce* parameter does not match the one in the ID token.

```
1  public Response validateTokens(... tokenResponse, ...oidcConfig) {
2      try {
3          IdToken idToken = tokenResponse.parseIdToken();
4          if(!oidcConfig.nonce.equals(idToken...getNonce())) {
5              return Response...UNAUTHORIZED...
6                      "Provided nonce did not match";
7          }
8          if(!idToken.verifySignature(publicKeyFromJwkSet())){
9              return Response.status(Response.Status.UNAUTHORIZED)
10                     "JWT signature is not valid";
11         }
12         ... other checks
13        ....createAndStoreCredential(tokenResponse, oidcConfig.appuuid);
14         return Response.ok()
15                 .entity(tokenResponse)
16     } catch (... | ... | ...Exception e) {
17         return Response.status(Response.Status.BAD_REQUEST).build();
18    }
19 }
```

Listing 6.11: Correct suggestion for token validation. The full example is found in Listing 5.4

Vulnerable code example

The basic error that is attempted covered with the CFG detector is cases where the developer makes a silly mistake and for instance forgets to enforce a check properly, or accidentally reverses a conditional. In the case in Listing 6.12 an if-check of the token signature is implemented in line 5, but the program flow is not broken. As a result, the program will continue down to line 9, where a success response is returned. In such a case, the program accidentally falls back on the green path even if the signatures did not match.

Another mistake developers may make is to write reversed logic. In line 1, the if-check is not checking for a negative value. As a result, the code will accept an incorrect issuer value, and an adversary impersonating an IdP might exploit this mistake.

```
1      if(idToken.verifyIssuer(...)){
2          // BUG: Reversed if conditional
3          return Response...UNAUTHORIZED...
4      }
5      if(!idToken.verifySignature(publicKey)){
6          // do something
7          // BUG: no return. Falls through to response OK.
8      }
9      ...
10     return Response.ok()
11             .entity(tokenResponse)
12             .build();
```

Listing 6.12: Respond incorrectly to a token verification.

Listing 6.13 shows other hypothesized ways the developer may write code that responds to checks with an incorrect control flow. In line 5, a return or throw statement that breaks the control flow is absent. In line 9 the developer accidentally returns a wrong response code, instead of an error code. Lastly, in line 14, the developer returns null. This is not necessarily a vulnerability, but is a code smell.

```
1  public Response validateToken(IdTokenResponse tokenResponse, OidcConfig
       oidcConfig) {
2      try {
3          IdToken idToken = tokenResponse.parseIdToken(); // Parse
4          if(!oidcConfig.nonce.equals(idToken.getPayload().getNonce())) {
5              // BUG: no return
6          }
7          ...
8          if(!idToken.verifyAudience(Collections.singleton(clientId))) {
9              return Response.ok().build();
10             // BUG: returns OK in wrong place
11         }
12         if(!idToken.verifyTime(Instant.now().toEpochMilli(),
13                                             DEFAULT_TIME_SKEW_SECONDS)){
14             return null;
15             // BUG: Smelly code returning null.
16         ...
17         ... createAndStoreCredential(tokenResponse, oidcConfig.appuuid);
18         return Response.ok()
19                 .entity(tokenResponse)
20                 .build();
21      } catch (... | ... | ...Exception e) {
22         return Response...BAD_REQUEST);
23     } catch (Exception e) {
24         return Response...INTERNAL_SERVER_ERROR);
25     }
26 }
```

Listing 6.13: Different ways to respond incorrectly to a token verification.

Vulnerability Patterns

Table 6.7 shows the two vulnerability patterns used by the Control flow ID token verification detector.

Table 6.7: Vulnerability patterns in the *Control flow ID token verification detector*

Pattern	Abbreviation	Description
IMPROPER_TOKEN_VERIFY _CONTROL_FLOW	SECITVCF	An if-check of one of the values in the ID token responds with an incorrect control flow.
REVERSED_IF_EQUALS _ID_TOKEN_VERIFY	SECREQTVER	An if-check expected to check for a negative boolean value checks for a positive one, possibly reversing the value it checks.

Detection strategy

This detector uses the Static Control Flow Check analysis, which is described in Section 6.2.3. It looks for patterns of ID token validation like the Improper ID token verification detector. It has the following outcomes when analyzing methods that do ID token verification:

- This basic block does one of the ID token checks.
- If this block does not have an `ifne` bytecode instruction, report REVERSED_IF_EQUALS_ID_TOKEN_VERIFY.
- If this basic block is not followed by a block that as a return instruction that is appropriate, report IMPROPER_TOKEN_VERIFY_CONTROL_FLOW.

Severity

If one of the checks does not have a correct response, or a conditional is reversed, the check is essentially useless. The relying party will then be at risk of a token forgery attack.

Chapter 7

Evaluation

This chapter contains a practical demonstration and validation of the analyses. This validation of the research results uses the “slice of life” examples taken from real code bases (See Chapter 4.3).

Section 7.1 goes through the experimental setup. Quantitative results and metrics are presented in tables in Section 7.2. Section 7.3 contains qualitative interpretation of the results, going into some key examples with insights into why the analysis yielded false positives or negatives, or succeeded.

7.1 Experimental setup

To validate the results of this thesis, the “slice of life” example-based strategy (See Section 4.3), with examples drawn from the field, is used as basis. It is also referred to as a quasi-experiment or field experiment [65, pp.133–134]. The essence of such experiments is trying to stay true to the spirit of laboratory experiments, but rather having focus on making observations of real-life settings, which make for a more naturally occurring experiment. Such a validation strategy is not as strong as more rigid statistically significant studies, but are fairly common for successful software engineering research [82].

The Example-based strategy is set in motion by using six open-source applications that implement code resembling OpenID Connect relying parties. The analyses in this thesis are highly specialized targeted at the OpenID Connect protocol flow, all code files in the applications waere not analyzed. Instead, the evaluation was limited to analyzing the files that contained relevant interfacing with the SDKs, and necessary boilerplate so that the relevant files would be altered as little as possible. Table 7.1 shows the open-source applications analyzed.

Table 7.1: Applications using OpenID Connect SDKs that were analyzed. Three using the Google library, and three using the Nimbus SDK. *Files* denoted files included from the given application.

Subject	Application	Description	SDK used	Files
ZopSpace [102]	zop-app *v0.1*	Video annotation Android app	Google	1
Atricore [7]	atricore-idbus *v1.4.3-27*	Identity federation platform	Google	10
Firebase [35]	firebase-admin-java *v6.13.0*	Admin SDK for connecting with Firebase services	Google	116
SonarQube [94]	sonar-auth-oidc *v2.0.0*	SonarQube SDK for authentication by Vaulttec	Nimbus	5
Liferay [52]	LiferayPortal *v7.3.2 GA3*	Business platform for Java	Nimbus	22
codice [14]	ddf, version *ddf-2.24.0*	Distributed Data Framework: modular integration framework	Nimbus	3

Like mentioned in Chapter 6.3, detectors have not been implemented for all the steps in the flow, and do not currently cover all the points in the checklist. For this demonstration it suffices to show a case for each type, though two detectors are implemented for Co-existing Invocation Enforcement. The detectors tested in the evaluation are shown in Table 7.2. The flow steps referred to in the table are explained in Chapter 5.1.

Table 7.2: Overview of the implemented analyses as Find Security Bugs detectors used in the evaluation.

Detector	Description	Type	Step in flow
State validation	Check callback after authentication request	Subsequent Inv.	2
ID Token validation	Check ID Token validation after token request	Subsequent Inv.	3
ID Token validation CFG	Check that the system responds properly to code validation checks	CFG Check	3
Authorization Grant	Check whether improper grants like Resource Owner Password Grant is used	Immed. Code Smell	2

The code bases used in the validation were obtained in Github open-source repositories on 18th of May 2020. Github repositories were searched for projects containing code that uses any of the two SDKs, the Nimbus SDK or the Google library, which are used as basis for the analyses in this thesis. The goal for the experiment on source code bases was to obtain three code bases using the Nimbus SDK, and three code bases using the Google library, aiming at a total of six open-source projects. The search protocol on Github was as follows:

- Extract identifiers of imports of SDKs that necessarily need to be included in OpenID Connect code that falls in the category of the analyses in this thesis. The query strings are as follows:
  - For the Nimbus SDK:

    ```
    import com.nimbusds.openid.connect.sdk
    ```

  - For the Google SDK:

    ```
    import com.google.api.client.auth.openidconnect.
    ```

- Perform a Github search with the given strings above, in code mode: `https://github.com/search?q=<query-string>&type=Code`. This mode searches the contents of code files.
- Scan through the first 20 pages for relevant code projects, or stop if goal is reached.
- The included code bases must satisfy the following inclusion criteria:
  1. This seems like a real code base used in a production setting, and is not just an example or a personal dummy project.
  2. The file that contains the code matching the search hit indeed seems to be implementing parts of, or the whole authorization code flow in a Relying Party.

The analyses were performed on a Lenovo P1 gen 2 running Ubuntu 19.04, with a Intel Core i7-9850H processor and a Graphics adapter NVIDIA Quadro T1000 (Laptop) 4096 MB and 32GB RAM.

Detectors were run in the FindBugs Command-Line Interface (CLI)[1], based on instructions in the CLI guide at FindSecBugs [73]. The tool was built in the following way from the root folder in the forked FindSecBugs project [28], in the `evaluation-opensource` branch[2]:

- `$ mvn clean install`
- `$ cd cli`
- `$ gradle assemble`
- `$ gradle package`

The analyzed applications were all cloned, and the files considered relevant were built to `.jar` files, because the CLI interface takes compiled java as `.jar` files for input. Inclusion criteria for the files was that they had imports of SDK classes that are used in OpenID Connect or OAuth, or that they were linked to classes that satisfied this constraint. As few files as possible to compile the given project without too much changes were included. Other files than this would not trigger detectors. The analyzed files were added to the Eval and build in the each their sub-module in the `Eval` project [29]. Then in the root folder of the

[1]Findbugs guide for running in command-line: `http://findbugs.sourceforge.net/manual/running.html`

[2]`evaluation-opensource` branch: `https://github.com/Eliassoren/find-sec-bugs/tree/feature/evaluation-opensource`

`Eval` project, `$ mvn clean install` was run to get a `.jar` file. The file was generated to `<submodule-subject>/target`.

Then in the FindSecBugs project, still in the `find-sec-bugs/cli` directory, each of the subjects were analyzed by entering the following command in the `find-sec-bugs/cli` directory:

```
$ ./findsecbugs.sh
    -output <subject>Results.xml
    -visitors ImproperTokenValidationDetector,
            TokenValidationCFGAnalysis,
            InsecureAuthorizationGrantDetector,
            MissingCheckStateOidcDetector
    <path from root>/Eval/<subject-module>/target/<subject>-1.0-SNAPSHOT.jar
```

The resulting analyses came as output in the named xml file in the `find-sec-bugs/cli` directory. This action runs the whole FindSecBugs plugin (also including FindBugs detectors) on the targeted code, including the analyses implemented in this thesis. Only the detectors that were triggered came in with specific analysis times in millisecond. The total time that is computed was the time for the whole plugin to run analyses on the given code. The files were reviewed manually to verify the analysis results.

The analyzed files are publicly available at the Oidc-FindSecbugs-Eval project on Github [29], and the detectors that were run in a forked FindSecBugs repository [28], in the branch *evaluation-opensource*. Most of the project files of the analyzed applications had to be altered because they had dependency errors or would not build from their master branch, or were hard to configure to run. These alterations mainly included commenting out internal library references, and did not touch any code artifacts likely to affect the analysis. The projects varied in size, and therefore required different inclusion volumes to compile without altering too much of the code. Most changes are documented with comments in the code in the Eval project.

The FindSecBugs plugin can normally be used in the IDE, but compatibility issues and an outdated version of FindSecBugs made it impossible to run in the IDE at this time. Users running the plugin in normal circumstances would simply install the plugin and run it targeted at their selected files, or on their whole project. That would make the process of running the analysis significantly easier than the during this evaluation.

7.1.1 Metrics for evaluation

To validate the performance of the analyses, false and true positives as well as false and true negatives were chosen, because they are common metrics in several publications. Based on these metrics, properties like *precision*, and *false positive rate* were selected for illustrating performances as fractions. *Recall* is also included, but this metric has lower confidence because the code analyzed is real applications, which means it is hard to get ground-truths about how many vulnerabilities exist in the applications. Still, some limited measure of the characteristic can be included to have some reference. The most important role of these characteristics will be to illustrate the performance of the analyses in a set of real-world cases.

To calculate the metrics, constraints have to be set for how positives and negatives are counted for the detectors. The following constraints are set for defining the evaluation metrics.

7.1.2 Definitions potential warnings for metrics

The d detectors used in the analysis each have a certain set of bug patterns P_i they may raise as warnings in a method. A class has a population m_{wr} methods which have the potential to be considered potential *warning-raising* methods. An application A has c classes.

Definition 7.1.1. Warning-raising methods Potential warning-raising methods, m_{wr}: methods in a class that imports the given SDK for OpenID connect, and executes an action which belongs to the universe in which the Detector does classification from, i.e. implements a part of OpenID Connect or OAuth 2.0. If the method for example delegates some of the verification checks to another method, both are considered relevant.

These methods provide a benchmark, and metrics are computed on only these potential warning-raising methods, while other boilerplate methods which would otherwise obscure the data are excluded. Timing of detectors also includes the other files, as the plugin scans the whole project.

Definition 7.1.2. Total potential warnings for a method The total amount of potential warnings W_M for a method, analyzed with d detectors each report reporting P_i patterns: $W_m = \sum_{i=1}^{d} P_i$

Definition 7.1.3. Potential warnings for a class The total amount of potential warnings W_C for a class: $W_C = m \cdot W_m$

Definition 7.1.4. Potential warnings for an application per detector The amount of potential warnings W_{AD} for an application with c classes for a detector with P patterns: $W_A = c \cdot m_{wr} \cdot P$

Definition 7.1.5. Total potential warnings for an application The total amount of potential warnings W_{AT} for an application with c classes: $W_{AT} = c \cdot C_w = c \cdot m_{wr} \cdot W_M = c \cdot m_{wr} \cdot \sum_{i=1}^{d} P_i$

True positives

The detector has a *true positive* if it raises warning in one of its rules is broken in a method, and the rule is in fact broken. Some warnings may link two methods since they relate to the same vulnerability, this is then counted as one positive. The set of true positives (TP) for a method is the amount of patterns P that are true rule violations.

False positives

The detector has a *false positive* (FP) if it raises warning in one of its rules is broken in a method, and the rule is followed in the code. A special case exists here. If the *ImproperTokenValidationDetector* notices that the code is missing four checks, it will raise a general warning plus each of the four instances. This is then counted as four reports. If all five checks it looks for are missing, it will condense them into one warning saying you miss five things. This is then counted as five positives, since the total absence of five properties raised this special warning. The set of false positives (FP) for a method is the amount of patterns P raised as warnings which are not rule violations.

True negatives

The detector has a *true negative* (TN) if it does not report any vulnerabilities, and the method in a class does not have a vulnerability breaking the rule. Each method satisfying this property is considered a true negative. The true negatives illustrate the difference between potentially raised warnings and true warnings. To further clarify, for any method where a detector can potentially report six different vulnerabilities, and none of these vulnerabilities exist in the method, the method had six true negatives. Based on Definition 7.1.5, the following definition is set for counting true negatives in the analysis results of an application:

Definition 7.1.6. Count true negatives method The total amount of true negatives TN_m for a method with a total of potential warnings W_m is defined:

$$TN_m = W_m - TP_m - FP_m - FN_m$$

Definition 7.1.7. Count true negatives application The total amount of true negatives TN_A for an application with a total of potential warnings W_{AT} and methods m is defined:

$$TN_A = W_{AT} - TP_A - FP_A - FN_A = \sum_{n=1}^{m} TN_m$$

False negatives

The detector has a *false negative* it it does not report the vulnerabilities in the code, but there is in fact a vulnerability in the code. The set of false negatives (FN are counted with manual review of the code, imitating the potential warnings that would have been raised by detectors.

7.2 Experimental results

This section contains the experimental results for the validation. Section 7.2.1 gives an overview of the numbers that go into the analyses, with results on each subject in Table 7.3, and a summary of the results in Table 7.5. Metrics based on these numbers are presented in Section 7.1.1. The raw data to the results on each subject are in Appendix B.

7.2.1 Overview

Table 7.3: Overview of the analysis results for the four detectors on each of the six applications. Subject: the application analyzed, Files: number of source files included in analysis, G: Google, N: Nimbus, t_t: total plugin clock run time in seconds, Detector: the implemented analysis, W_A: the potential number of warnings for a detector analyzing an application, TP: True positives (vulnerabilities found), FP: false positives, FN: false negatives, TN: true negatives, t: run time as clock milliseconds,

Subject	SDK	t_t(s)	Detector	W_A	TP	FP	TN	FN
ZopSpace	G	0.79	Auth. Gr.	1	0	0	1	0
			State ver.	2	0	0	1	1
			Token ver.	21	9	0	12	0
			Token CFG	0	0	0	0	0
Atricore	G	1.06	Auth. Gr	1	0	0	1	0
			State ver.	2	0	0	2	0
			Token ver.	5	5	0	2	0
			Token CFG	0	0	0	0	0
Firebase	G	2.82	Auth. Gr	2	0	0	1	0
			State ver.	0	0	0	0	0
			Token ver.	56	1	3	52	0
			Token CFG	10	0	4	6	0
SonarQube	N	23.1	Auth. Gr	2	0	0	2	0
			State ver.	4	0	1	0	0
			Token ver.	21	5	5	11	0
			Token CFG	0	0	0	0	0
Liferay	N	1.38	Auth. Gr	2	0	0	2	0
			State ver.	6	0	1	0	0
			Token ver.	21	0	0	21	0
			Token CFG	0	0	0	0	0
codice	N	1.09	Auth. Gr	1	0	0	1	0
			State ver.	0	0	0	1	0
			Token ver.	28	0	5	23	0
			Token CFG	0	0	0	0	0

Table 7.4: Total of analysis results for detectors. W_{AT}: the total potential number of warnings a detector may raise, TP: True positives (vulnerabilities found), FP: false positives, FN: false negatives, TN: true negatives

Detector	W_{AT}	TP	FP	TN	FN
Auth.Gr	9	0	0	9	0
State ver.	14	0	2	10	1
Token ver.	154	20	13	121	0
Token CFG	10	0	4	6	0
SUM	187	20	19	147	1

Like Table 7.5 shows, it is the *Co-existing Invocation Enforcement* analysis implemented through the *ImproperTokenValidationDetector* that got the highest volume of detected vulnerabilities, and had the most relevant analysis points in W_A.

Table 7.5: Analysis results for the performance of the detectors, grouped per SDK. W_{AT}: the total potential number of warnings a detector may raise in a given application, TP: True positives (vulnerabilities found), FP: false positives, FN: false negatives, TN: true negatives.

SDK	Detector	W_{AT}	TP	FP	TN	FN
Google	Auth.Gr	4	0	0	9	0
	State ver.	4	0	0	3	1
	Token ver.	84	15	3	66	0
	Token CFG	10	0	4	6	0
	SUM G	102	15	7	79	1
Nimbus	Auth.Gr	5	0	0	5	0
	State ver.	10	0	2	8	0
	Token ver.	70	5	10	55	0
	Token CFG	0	0	0	0	0
	SUM N	85	5	12	68	0
	SUM	187	20	19	147	1

7.2.2 Metrics

From the numbers of the analyses, the metrics Precision, Recall and True Negative Rate (TNR) were computed. These are explained and defined in Chapter 2.6.

$$Precision = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

$$Recall = \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}}$$

$$\textit{True negative rate} = \frac{\text{True negatives}}{\text{False positives} + \text{True negatives}}$$

When looking at all the detectors on all the code files, the evaluation yielded an unsatisfactory precision of 51%. With further investigation, this may be caused by certain outliers in a subset of the analyzed applications. Two of the applications implementing the Nimbus SDK had a code structure that lead to false positives. These outliers likely skewed the data. Insights into these false positives are explained in Sections 7.3. When looking at the data of the Token ver. detector for the applications in the Google API, it has a promising precision of 83%.

Table 7.6 shows the metrics for the detectors. The metrics for the results analyzing the two different groups of applications are shown. One of the groups contains the applications using the Google library, while the other contains the ones using the Nimbus SDK. There were some internal differences when looking at these two groups. The total metrics for the analysis of all the analyzed subjects is shown in the bottom of the table.

Table 7.6: Metrics for analysis results per detector and total: Precision, Recall and True Negative Rate (TNR). The totals for Google (Total G) and Nimbus (Total N) are calculated using the complete numbers in SUM G and SUM N in Table 7.5, including numbers from all the detectors.

SDK	Detector	Precision	Recall	TNR
Google	Auth. Gr			1
	State ver.			1
	Token ver.	0.83	1	0.96
	Token CFG			0.6
	Total G	0.68	0.93	0.92
Nimbus	Auth. Gr			1
	State ver.			0.8
	Token ver.	0.33	1	0.85
	Token CFG			0.6
	Total N	0.29	1	0.85
BOTH	Auth. Gr			1
	State ver.			0.84
	Token ver.	0.61	1	0.90
	Token CFG			0.6
	Total	0.51	0.95	0.89

7.3 Qualitative analysis

This section goes through some qualitative analysis, interpreting some reasons for the outcomes. Sections 7.3.1 to 7.3.3 contain analysis insights in the subjects using the Google library, while Sections 7.3.4 to 7.3.6 go through the analyzed subjects using the Nimbus SDK.

Emphasis is put on Improper ID token verification detector, because was the detector with the largest volume of data, and the only detector with true positives.

7.3.1 ZopSpace

ZopSpace is one of the google-library using applications, and is as the only one, an Android app [102]. This heavily limited the inclusion of files, since Android SDKs will not easily compile in an ordinary Java run-time environment. However the one relevant file was mostly written in "ordinary" Java, and required few changes to make it compile.

When analyzing this code base, the *ID token validation* and partly the *State validation* detector yielded the most interesting results, which illustrate both strengths and weaknesses with the analyses.

MissingCheckStateOidcDetector

The *MissingCheckStateOidcDetector* looking for improper or missing validation of the *state* parameter failed to pick up that this was missing in the code in this app. It only had one file to analyze from, but the expected patterns it is currently set to look for did not exist in this code (The patterns it looks for are covered in Chapter 6.4.2).

This means that CSRF protection is not implemented, and manual searches through the code base found no signs of *state* parameter any other place in the Android-parts of the code base either. Listing 7.3 shows the code snippet which contains code that reveals the absence of the state parameter. In lines 5-13, the authentication request is build. The *state* parameter must be included here.

```
1  String newAuthorizationUrl(String authorizationServerUrl,...
2                             String[] scopes) {
3
4          List<String> scopesList = Arrays.asList(scopes);
5          AuthorizationCodeFlow flow = new AuthorizationCodeFlow.Builder(
6                  BearerToken.authorizationHeaderAccessMethod(),
7                  new NetHttpTransport(),
8                  new GsonFactory(),
9                  new GenericUrl(tokenServerUrl),
10                 new BasicAuthentication(clientId, clientSecret),
11                 clientId,
12                 authorizationServerUrl
13         ).build();
14         AuthorizationCodeRequestUrl authUrl = flow.newAuthorizationUrl()
    ;
15         authUrl.setScopes(scopesList);
16         authUrl.setRedirectUri(redirectUrl);
17         if (scopesList.contains("offline_access")) {
18             authUrl.set("prompt", "consent");
19         }
20         authUrl.set("prompt", "login");
21         authUrl.set("display", "touch");
22         return authUrl.toString();
23     }
```

Listing 7.1: Step 1 in zopspace, building the Authorization Code request URL. The URL clearly misses addition of the *state* parameter.

This absence was not picked up by the MissingCheckStateOidcDetector, which uses the callback context as its entry-point trigger for analysis. However another detector could be added, which focuses on how the request URL is build in step 1. Such a detector is suggested in Table 6.2. It could use the following process to detect the bug above:

1. Look for signs of instantiating an *AuthorizationCodeRequestUrl* type.
2. Look for the `setState()` method call on the request url object (which comes in some form to allow adding the state parameter).

If the first point is satisfied, the second one must also be satisfied. Otherwise a warning of missing state parameter is raised.

ImproperTokenValidationDetector

The Improper ID token verification detector was able to pick up 10 vulnerabilities of incomplete validation of the ID token in the Zop-app project. It had four true positives which were just as expected, in the *isValidIdToken* method in Listing 7.2. In line 6, the IdTokenVerifier is instantiated, and the audience parameter is added. When `verifier.verify()` is called in line 10, only the *freshness* and audience parameter is verified. The analysis correctly reported that this file misses validation of the nonce, signature and iss parameters. Additionally it reported USING_INCOMPLETE_ID_TOKEN_VALIDATOR, which is a vulnerability pattern reported when the verifier is implemented incompletely.

```
1  boolean isValidIdToken(String clientId, String tokenString) {
2      if (clientId == null || tokenString == null) {
3          return false;
4      }
5      List<String> audiences = Collections.singletonList(clientId);
6      IdTokenVerifier verifier = new IdTokenVerifier
7                                      .Builder()
8                                      .setAudience(audiences).build();
9      IdToken idToken = IdToken.parse(new GsonFactory(), tokenString);
10     return verifier.verify(idToken);
11 }
```

Listing 7.2: Incomplete ID Token verification in an open-source android app project [102].

Interestingly, the analysis also unexpectedly picked up five true positives in `refreshTokens`, even though this is currently not covered by the model. This was picked up because the detector looks for *token requests* as pattern A, and then expects validation as B. This is however also a true vulnerability according to the OpenID Connect specification [60], since it is required to perform a complete ID token verification if an ID token is included in the refresh request. The example here shows that the analysis is applicable, even if its initial design was based on a more restricted mental model.

```

IdTokenResponse refreshTokens(String tokenServerUrl,...,
                              String refreshToken) {
    List<String> scopesList = Arrays.asList(scopes);
    RefreshTokenRequest request = new RefreshTokenRequest(
            new NetHttpTransport(),
            new GsonFactory(),
            new GenericUrl(tokenServerUrl),
            refreshToken
    );
    request.setClientAuthentication(...clientId, clientSecret);
    request.setScopes(scopesList);
    return IdTokenResponse.execute(request);
}
```

Listing 7.3: Refresh token request. Currently not covered by the model in this thesis, but the analysis still picked it up.

7.3.2 Atricore

Atricode-idbus, the Atricore Identity Bus Platform [7] has one file which was considered relevant upon inspection, the *GoogleAuthzTokenConsumerProducer* which uses the Google library. Due to their complex architecture, several other files touch upon OpenID Connect messages. These are not included, both since their relevance is limited, and because it would require a lot of work to make these files compile.

Token validation is clearly missing in the method *doProcessAuthzTokenResponse* in Listing 7.4. After retrieving the ID token in, the code directly proceeds to retrieve user info.

```
protected void doProcessAuthzTokenResponse(CamelMediationExchange
   exchange) {
    ... correct authorization response parsing
    ... state validation
    request.setRedirectUri(accessTokenConsumerLocation.getLocation());
    IdTokenResponse idTokenResponse = (IdTokenResponse) mediator.
   sendMessage(request, accessTokenConsumerLocation);
    IdToken idToken = idTokenResponse.parseIdToken();
    // NO ID token validation!!
    ... userinfo request
}
```

Listing 7.4: Steps 2-3 in atricore, token request. Token validation is missing.

This turned out to be a case which worked exactly as expected for the *ImproperToken-ValidationDetector*. The trigger was a call to `idTokenResponse.parseIdToken()`. Then the detector expected to find validation. Failing to find this in the method, it raised warning that all five required checks are missing.

7.3.3 Firebase

Firebase is an open-source app development platform delivered by Google. The code analyzed is their admin Java SDK [35]. Different from the other code bases, this application turns out to not implement the entire OpenID Connect protocol. Rather, they have they own way of doing authorization and authentication, but are using the types of the Google OIDC library to wrap around their JSON Web Tokens, which are used to manage identities. This strictly means that the discovery here is not a true bug, given that they do not follow the protocol specification in the first place, and therefore are not bound by its rules. However given the rule set of the detectors, the broken rule of missing nonce was correctly identified in the code.

Improper ID token verification detector

Listing 7.5 contains a method which provoked three false positives in Firebase. The method returns the IdTokenVerifier class in the Google Library, which does not completely do all the required validation checks. The detector assumes that this method is used to validate the ID token, and correctly identifies that this method in itself misses the signature validation and nonce validation checks. However, this method is called elsewhere, and its result is used in a place where the needed checks are in fact implemented.

The method in Listing 7.5 is implemented in an entirely different code file than the other relevant code, and effectively dodges the inherent single-file analysis design of FindBugs, which the analyses in this thesis is limited to. These false positives could be avoided by modifying the analysis to set stricter rules for the conditions that form an entry-pattern. However, imposing such stricter rules would also yield some false negatives, in a case.

```java
private static IdTokenVerifier newIdTokenVerifier(Clock clock,
                                                   String issuerPrefix,
                                                   String projectId) {
    return new IdTokenVerifier.Builder()
        .setClock(clock)
        .setAudience(ImmutableList.of(projectId))
        .setIssuer(issuerPrefix + projectId)
        .build();
}
```

Listing 7.5: A method which provoked three false positives in Firebase.

Control flow ID token verification detector

Firebase was the only application implementing code where the Control flow ID token verification detector was relevant. It yielded four false positives.

One of the false positives was the of the REVERSED_IF_EQUALS_ID_TOKEN_VERIFY pattern, shown in method `isSignatureValid` in Listing 7.6. The check on line 3 is reversed from what is expected in the model that drives the Control flow ID token verification detector, which wants to see checks like `!isValidABC()`. Such a negative check does indeed come in the method that calls this method, shown in Listing 7.7 in line 3. Cases like this can probably be accounted for if the analysis is re-engineered slightly to reason about inter-procedural heuristics like the Co-existing Invocation Enforcement analysis does.

```
1 private boolean isSignatureValid(IdToken token) throws
    GeneralSecurityException, IOException {
2     for (PublicKey key : publicKeysManager.getPublicKeys()) {
3       if (token.verifySignature(key)) {
4         return true;
5       }
6     }
7     return false;
8   }
```

Listing 7.6: Method in Firebase that raised reversed if-conditional check.

```
private void checkSignature(IdToken token) throws ExportedUserRecord.
    FirebaseAuthException {
    try {
      if (!isSignatureValid(token)) {
        throw new ExportedUserRecord.FirebaseAuthException(
    ERROR_INVALID_CREDENTIAL,
                String.format(
                    "Failed to verify the signature of Firebase %s. %s",
                    shortName,
                    getVerifyTokenMessage()));
      }
      ...
```

Listing 7.7: Method in Firebase that raised reversed if conditional check warning.

The other three false positives were of the pattern IMPROPER_TOKEN_VERIFY_CONTROL_FLOW. The idea of that vulnerability pattern is based on the developer-oriented model, which states that a HTTP 401 should be returned after such a failed check. If an exception is thrown this is also accepted. This turned out to be too a narrow scope for the model. The method in Listing 7.6 gives a warning because `true` is returned after the check, in line 4.

Listing 7.8 shows a validation method that raised the remaining two warnings. Two ID token checks like the one in line 6 were identified in the method, both raising the same warning. Instead of directly returning an error with a message, an error message is appended to a collector string. The validation method is called by the method in Listing 7.9. Here if any check has appended something to the error message, an exception is thrown in line 5. This code structure completely evades what was intended in the design of the Control flow ID token verification detector. The false positives may possibly be evaded if it gets some more patterns. Some substantial engineering work is however needed to make it account for this structure.

```
1  private String getErrorIfContentInvalid(final IdToken idToken) {
2      final Header header = idToken.getHeader();
3      final Payload payload = idToken.getPayload();
4      String errorMessage = null;
5      ...
6      else if (!idToken.verifyAudience(idTokenVerifier.getAudience())) {
7        errorMessage = String.format(
8            "Firebase %s has incorrect ...
9            joinWithComma(idTokenVerifier.getAudience())...
10     }
```

```
    ...
```

Listing 7.8: Method in Firebase that raised improper control flow warning.

```
private void checkContents(final IdToken token) {
    String errorMessage = getErrorIfContentInvalid(token);
    if (errorMessage != null) {
      String detailedError = String.format("%s %s", errorMessage,
   getVerifyTokenMessage());
      throw new ExportedUserRecord.FirebaseAuthException(
   ERROR_INVALID_CREDENTIAL, detailedError);
    }
}
```

Listing 7.9: Method in Firebase that raised improper control flow warning.

7.3.4 SonarQube

Improper state verification detector

In the admin api of SonarQube [94], the Improper state verification detector yielded a false positive. This false positive is completely impossible for a tool like FindSecBugs to detect. Listing 7.10 contains the step 2 callback method called `getAuthorizationCode`. The parse call in line 5 is the trigger of the analysis. However the state parameter is not verified here. Instead, it is verified in another class, in the method shown in Listing 7.11 in line 2. It verifies the value passively, and we cannot reason about the type of the State parameter at all. The way these have implemented their code, it is impossible for the analysis to avoid the false positive. FindBugs detectors inherently cannot reason about facts that cross Java classes, since they visit one and one class.

```
1  public AuthorizationCode getAuthorizationCode(HttpServletRequest
      callbackRequest) {
2      AuthenticationResponse authResponse = null;
3      try {
4        HTTPRequest request = ServletUtils.createHTTPRequest(
      callbackRequest);
5        authResponse = AuthenticationResponseParser.parse(request.getURL()
      .toURI(), request.getQueryParameters());
6      } catch (ParseException | URISyntaxException | IOException e) {
7        throw new IllegalStateException("Error while parsing callback
      request", e);
8      }
9      if (authResponse instanceof AuthenticationErrorResponse) {
10       ErrorObject error = ((AuthenticationErrorResponse) authResponse).
      getErrorObject();
11       throw new IllegalStateException("Authentication request failed: "
      + error.toJSONObject());
12     }
13     AuthorizationCode authorizationCode = ((
      AuthenticationSuccessResponse) authResponse).getAuthorizationCode();
14     return authorizationCode;
15   }
16 }
```

Listing 7.10: Callback method in SonarQube yielding a false positive for the Improper state verification detector.

```java
public void callback(CallbackContext context) {
     context.verifyCsrfState();
     AuthorizationCode authorizationCode = client.getAuthorizationCode(
    context.getRequest());
     UserInfo userInfo = client.getUserInfo(authorizationCode, context.
    getCallbackUrl());
     UserIdentity userIdentity = userIdentityFactory.create(userInfo);
     userIdentity.getGroups());
     context.authenticate(userIdentity);
     context.redirectToRequestedPage();
}
```

Listing 7.11: State verification happening in another class in SonarQube.

Improper ID token verification detector

Listing 7.12 shows the method that yielded five false positives in SonarQube. The analysis incorrectly thinks this method needs five checks, because it sends token request and simply passes its result with `OIDCTokenResponseParser.parse(response)` in line 7. It is however not this method that is required to actually have checks, but rather the method that calls this method. In this case, the method that called this one also missed the checks, and had five true positives.

Still it is incorrect to flag `getTokenResponse` as a vulnerable method. However, like explained in Section 7.3.6, a case like this yields false positives even in the code correctly implements checks elsewhere.

```java
1   protected TokenResponse getTokenResponse(AuthorizationCode
      authorizationCode, String callbackUrl) {
2      try {
3        URI tokenEndpointURI = getProviderMetadata().getTokenEndpointURI()
      ;
4        TokenRequest request = new TokenRequest(tokenEndpointURI, new
      ClientSecretBasic(getClientId(), getClientSecret()),
5            new AuthorizationCodeGrant(authorizationCode, new URI(
      callbackUrl)));
6        HTTPResponse response = request.toHTTPRequest().send();
7        return OIDCTokenResponseParser.parse(response);
8      } catch (URISyntaxException | ParseException e) {
9        throw new IllegalStateException("Retrieving access token failed",
      e);
10     } catch (IOException e) {
11       throw new IllegalStateException("Retrieving access token failed: "
12           + "Identity provider not reachable - check network proxy
      setting 'http.nonProxyHosts' in 'sonar.properties'");
13     }
14   }
```

Listing 7.12: A method that yielded five false positive in SonarQube.

7.3.5 Liferay

In the Liferay portal [52], the Improper state verification detector gave a false positive. This false positive came for the same reason as for why the token validation gets a false positive in Codice (Section 7.3.6).

The validation of the *state* parameter happens in a different method which calls the method triggering the analysis, and it is not possible to eliminate this false positive without the same effort described in Section 7.3.6. This is an inherent limitation of the simple detector-baser analyses in FindBugs, which visits each class individually.

7.3.6 Codice

Listing 7.13 shows the method that yielded five false positive in Codice [14]. The analysis incorrectly thinks this method needs five checks, because it sends token request and simply passes its result in the end line where it returns `tokenSuccessResponse.getOIDCTokens()`. It is however not this location that is required to actually have checks, but the method that calls this one.

```java
public static OIDCTokens getOidcTokens(
    AuthorizationGrant grant,
    OIDCProviderMetadata metadata,
    ClientAuthentication clientAuthentication,
    int connectTimeout,
    int readTimeout)
    throws IOException, ParseException {
  final TokenRequest request =
      new TokenRequest(metadata.getTokenEndpointURI(),
 clientAuthentication, grant);
  HTTPRequest tokenHttpRequest = request.toHTTPRequest();
  tokenHttpRequest.setConnectTimeout(connectTimeout);
  tokenHttpRequest.setReadTimeout(readTimeout);
  final HTTPResponse httpResponse = tokenHttpRequest.send();
  LOGGER.debug(
      "Token response: status={}, content={}",
      httpResponse.getStatusCode(),
      httpResponse.getContent());
  final TokenResponse response = OIDCTokenResponseParser.parse(
 httpResponse);
  if (response instanceof TokenErrorResponse) {
    throw new TechnicalException(
        "Bad token response, error=" + ((TokenErrorResponse) response)
 .getErrorObject());
  }
  LOGGER.debug("Token response successful");
  final OIDCTokenResponse tokenSuccessResponse = (OIDCTokenResponse)
 response;
  return tokenSuccessResponse.getOIDCTokens();
}
```

Listing 7.13: A method that yielded five false positive in Codice/ddf [14].

The method that called this one was not flagged for warnings, as it passed the rules of the analysis. This this method, which is called by the other one, gets positives. The reason for this is a limitation on how the Co-existing Invocation Enforcement analysis handles inter-procedural heuristics. The Co-existing Invocation Enforcement uses simple heuristics to approximate one piece of inter-procedural analysis, by going *down* in the call graph. This was modeled for the case where a certain check is delegated down to another method. Code structured like the pseudo code below will *not* give false positives, because the method `verifyB()` contains the needed checks, which is then called by method `getA()`. Some instruction `trigger()` will alert the detector that a set of checks is required:

```
boolean verifyB(data) {
    if check data.a;
    if check data.b;
    if check data....
}
Response getA() {
   trigger() // a trigger that says checks are needed
   if verifyB(data)
  ...
  return response;
}
```

This is solved, and the initial false positive is avoided because we "put aside" `verifyB()` and `getA()`, and do a double check after the initial screening analysis is done.`getA()` is then cleared from being a potentially vulnerable method when it is verified that it is assosciated with `verifyB()`, which contains the required checks. This is explained further in Chapter 6.2.2.

However in the case for this set of false positives, the code has the opposite structure, essentially requiring the analysis to be able to go upwards in the call graph. The “level” above this current method, verification might actually happen. The code like below will give false positives:

```
Trigger getTrigger(data) {
    ...
    return trigger();
  // Here the detector expected some checks!
}


Response getA() {
   Trigger t = getTrigger();
```

```
    if check a;
    if check b;
    if check...;
    return response;
}
```

The detector notices that the method `getA()` contains checks, and will correctly say that it is safe. Still the instruction `trigger()` alerts the detector that some checks are needed to follow the rules. However this would be a reversed association between the methods, compared to the heuristic that is used in the other example above. The principle for detecting that these are in fact negatives could be similar, and another set of associated methods could possibly be put aside and checked.

To do this however, the whole analysis would need a redesign since it is inherently intra-procedural with a simple inter-procedural check. This requires a significant amount of time, and it might take a full month to properly extend the analysis to account for this. The "downward" checks are just based on another linear verification of the code, and still keeps the detector rather simple. The same might be done for the above code snippet, but this may also have other effects on the recall of the analysis, and introduces complexity to the detector.

It would then have to approximate what happens both upwards and downwards in the call graph with only simple heuristics. To get to this level one might as well just have to compute a call graph, which introduces a different level of complexity, and goes beyond the scope of the simple nature of the Co-existing Invocation Enforcement analysis.

Chapter 8

Discussion

8.1 RQ1: The developer-oriented model

This section discusses results related to RQ1: *What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?*

The developer-oriented model of OpenID Connect proposed in this master thesis might lay a foundation for two things. Firstly, it may be a step in the direction for a rather general developer guide, but with specific, concrete steps in a checklist. Secondly, it may form guidelines for making more static analyses for usage of SDKs.

The qualitative model presented in this thesis is still a rough one, and does not entail all the steps that necessarily would go into the flow (refresh tokens and UserInfo requests have not yet been covered). With further refinement it would be interesting to see whether it could serve as a complete "angled" part of the specification, which communicates more clearly what to do by collating the essential information for the developer, and leaving the rest out. It would also be interesting to see whether such a model could be developed for the counterparts, namely for the ones developing an IdP. Also here static analyses could likely be employed to help enforce the rules.

What can be considered the main feature with such a model is to get a pragmatic, clear and unambiguous mental model, which is explicit and shared between a vulnerability testing tool and the developer. This way, as a contrary to the flexibility of the original specification, a more clear and restricted rule set can be named. Even though many security-enhancing values and parameters are named as optional in the specification, it can be argued that a stricter rule set could at least be presented for the developers. As long as it does not severely restrict *availability*, the general recommendations and requirements in OpenID Connect should be more focused on enhancing *confidentiality* and *integrity*. This is especially relevant since the protocol is employed in more and more systems with its dominance in the industry rising.

The specificity of the model proposed in this thesis can be considered both an advantage and a weakness. On one side, the specificity allows us to get a more clear set of rules to enforce when performing automated analyses like the ones designed and implemented in this thesis. The rules are rooted in explicit code examples, and the specific SDKs are likely to be used by many applications. Hence by adding rules for more SDKs, a significant part of the client applications on the web can be secured, as long as they use a popular and well-known SDK to support their implementation. However the specificity also introduces some weakness, as it can have a too narrow view for some applications, and may miss some relevant vulnerabilities. There are examples of this shown in the evaluation of the implemented static analyses (Chapter 7.3). The restrictions of the model makes the analyses prone to both false positives and false negatives. However it is shown by the demonstration that analyses based on the model found vulnerabilities in real-world code.

8.1.1 Comparison to related work

The existing security analyses of OpenID Connect and OAuth 2.0 elaborated in Chapter 3.1 can be described by a few common attributes: they are comprehensive and formal, they look at the overall communication between the entities, and they tend to be attacker-oriented. Being attacker-oriented means that their abstraction and view on the protocol is through a certain set of eyes, from an attacker standing on the outside of the parts in the system. In contrast, the goal of the model in this thesis is to provide an internal developer-view perspective, which is intended to be simple and pragmatic. This may serve to complement the comprehensive “bird-perspective” models that are generally used or presented in security analyses, and the specification itself.

A potential challenge with existing formal security analyses [33, 34, 62, 89] and models for OIDC is that they seek to be complete, involving all the entities and steps in a comprehensive model of the protocol. While such analyses and models definitely are important knowledge frameworks, their complexity limits their relevance to ordinary software developers who just want to add their app as a Relying Party.

Other relevant analyses are more practically oriented, looking at implementations of the protocol [2, 49, 50, 86]. However, they still have the attacker-oriented perspective, and have a rather formal approach to their analysis. Their advantage is that they are based on true implementations, which may make it easier to relate to and comprehend for a developer.

Since OIDC is distributed, its complete security is arguably a shared responsibility between the developer of the SDKs, the developer of the IdP and the developer of the Relying Party. Still it is not necessarily the RP developer’s concern to know in detail what the IdP does on the other side. These developers could instead use a straight-forward and simplified protocol model with a checklist of what concrete steps they need to take to ensure security on *their* end, like the model presented in this thesis. Beside the comprehensive models and the official specification, SDKs also provide developer guides which give clear and simple instructions to the developer. Their instructions alone do however not give a comprehensive checklist for security, but are rather focused on the simplest way of setting up a working flow.

When looking at development, implementation and usage of OpenID Connect, we can name several roles. In the development stage the entities of Identity Provider and Relying Party must be developed.

The IdP is often delivered as a service or product by a large organization, and is likely to be put under a careful testing regime, which enhances their security. The IdP organizations often also provide their own type sets or development kits that ordinary developers can use to integrate their app with the IdP. Nevertheless, developers implementing an their app are not unlikely to fail in writing some critical implementation details correctly. It is apparent by the analyses of the two SDKs and their developer guides [15, 19, 37], that the SDKs themselves not necessarily give all the answers regarding security practices either. Instead they often have very simple "get-started" guides, and have limited security guidelines compared to the protocol specification.

The steps needed to securely implement a Relying Party or any other protocol entity are thoroughly explained in the official specification [60]. The SDKs provide developer guides [19, 37] showing simple code samples for what is needed to get started in implementing the protocol flow. Even together, the specification and guides may be confusing sources as the official specification has loads of information, while the developer guides are rather short and simple.

Analysis of these SDKs and the specification can be extracted to a qualitative model of the flow containing a few concrete and simple rules for *secure* development of an RP, formed as a straightforward *checklist*. Even with such rules at hand, it could be bothersome or hard for a non-security competent developer to realize the gravity of failing to implement a certain security feature. Developers are also prone to forgetting a certain check or performing a check in the wrong way. Therefore in addition to serve as a guideline for the developers, this model forms a knowledge framework for the simple static analyses, which in their turn help enforce the rules of the protocol.

8.2 RQ2: Implementation of static analysis

This section relates to RQ2: How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?.

8.2.1 Experimental results of the detectors

In this study, four detectors were implemented to test the effectiveness of the analysis techniques. The validation experiment on six open-source applications showed that these techniques can find vulnerabilities in real code bases. The *Improper ID token verification detector* [1] found 20 vulnerabilities related to improper validation of ID tokens in the six open-source applications.

[1]The Improper ID token verification detector (Chapter 6.4.3 is an implementation of the *Co-existing Invocation Enforcement* analysis, which is explained in Chapter 6.2.2.

The *recall* of the tool in total was 95%, meaning most of the known vulnerable code was discovered. The tool also had a *precision* of 51% in total, and 61% for the Improper ID token verification detector as the only detector finding vulnerabilities. The recall looks promising, but this precision might look unsatisfactory low at first sight, since developers generally like precise tools [13]. While the analysis has some weaknesses in today's implementation, there are several arguments for why this metric should not alone be taken as a definitive measure of the strength and potential of the tool.

Firstly, the population of the applications analyzed is not statistically significant, meaning the result in total may be caused by a coincidence. If the precision had turned out to be for instance, 85% for these applications, this is still not a metric that on its own could confidently describe the performance of the tool. The data from the validation are arguably more nuanced than that, and the qualitative insights are important to determine its potential strengths and weaknesses in a small data. The only test characteristic that is fully satisfied by this case study is *realism* [26].

Three out of four detectors had no positives, and this itself might skew the results. Additionally, a significant portion (50%) of the FPs came due to one factor in the code structure of two applications. This is considered an outlier that greatly impacted the small data set, skewing the results. It can be argued that the outlier comes due to an overly complex code structure [2], which is an anti-pattern in security code [3].

The analysis was on a contrary rather effective on the code bases that implemented OIDC in a way that is more similar to the developer guides given by the SDKs [19, 37]. This makes sense since the analyses are based on the developer-oriented model, which is greatly influenced by these guides.

The Improper ID token verification detector has a precision of 61% in total. On a closer look, if disregarding the outlier, a promising 83% precision would be found for the Improper ID token verification detector. Removed outliers would leave only three false positives than cannot be dealt with [3]. This would be in the range of what is considered acceptable for most developers, which generally are not likely to accept a precision of any lower than 80% [13, 88].

With an intuitive reasoning from the qualitative analysis in Chapter 7.3, probably 15 of the 20 false positives in this trial can be avoided with improvements of the details in the analysis. Doing this requires a substantial engineering effort. It may take a few week's work to fix without sacrificing recall. This was not possible in the time constraints of this thesis.

Instead of concluding directly based on precision and recall, however, it is more valuable to also emphasize the qualitative insights of the results [4].

The metrics should also be interpreted based on the limitations of the data. Precision and recall are sensitive to imbalanced data sets [90]. In a small data set it is more useful to look at the *true negative rate* (TNR) together with the precision and recall. While the precision and recall can change significantly if exposed to outliers in small data sets, the true negative

[2]Outliers are described in detail in Chapters 7.3.4 and 7.3.6

[3]The false positives related to ID token verification that cannot be fixed are explained in Chapter 7.3.3.

[4]Qualitative insights into the details of the analyses are presented in Chapter 7.3.

rate is not affected as much by coincidences. In contrast to the other two metrics, the true negative rate is robust when facing imbalanced data [90].

The true negative rate was of 90%, meaning that 9 out of 10 non-vulnerable cases were correctly predicted as negatives. This gives a more nuanced picture of the resistance to false positives, and shows that in most cases, the analysis avoids giving false positives. The recall is only an indicative metric which is more confidently calculated in controlled test suites [5] More empirical testing is however needed to learn more about how strongly the analyses truly can perform, as all the metrics are affected by the size of the data set.

The Control flow ID token verification detector performed significantly better on the applications using the Google library than the ones using the Nimbus SDK, and 15 of the 20 discovered vulnerabilities were found in applications using the Google library. Additionally it had significantly different results for the metrics when looking at these two groups of applications. Table 7.6 shows that precision analyzing the applications that use the Google library, was at 68% for the current analysis.

Finally, if the total precision in should in fact turn out to be of 51% in the end, even this might be acceptable because the code analyzed is security-critical. Sørensen et al. [88] found that developers are more interested in finding all the vulnerabilities than having a high precision when looking at security-critical code.

But as it stands, the precision of the tool as implemented today is not likely to be satisfactory for a general practitioner. However the precision can be increased significantly through a few weeks of careful engineering efforts. This is an interesting avenue for further work.

FindSecBugs has been found to be quite precise in detection of several of the OWASP top 10 vulnerabilities [48]. However most of such the existing detectors (especially the ones which use similar techniques to the Improper ID token verification detector) can effectively be fooled into yielding false positives if the logic of the methods is structured in a certain way, similar to like the outliers found in this study. This must be considered since static analysis is dependent on the code structure.

The detectors are implemented using FindSecBugs, which has been found to have a superior usability to other known tools [48]. FindSecBugs easily integrates in the workflow of the developer, which increases the chance that developers will use these analyses to eliminate vulnerabilities [88]. Before this, no detectors for OpenID Connect vulnerabilities have been implemented in FindSecBugs, highlighting the novelty of this work.

The other three detectors

The three other detectors gave no true positives, and thereby did not generate much data to analyze. This might be because they are narrow, or because it is not as common for developers to make mistakes introducing these vulnerabilities in their code.

The Insecure authorization grant detector has a very limited scope, and correctly predicted 9 true negatives. It is not the most important detector, but contributes to avoiding special code smells. Also the Control flow ID token verification detector has a very special-

[5] See Chapter 2.6.3 about test suites.

ized scope, and only one of the six applications had relevant code i predicted. It had four false positives, three of which are possible to avoid if it is extended to look for a few more patterns. To eliminate the final false positive, it may have to get some inter-procedural attributes similar to the ones in the Co-existing Invocation Enforcement analysis.

Meanwhile the Improper state verification detector had a total of two false positives, yielding a true negative rate of 84%. The two false positives came due to the code structure in the analyzed projects. One of these cases cannot be avoided because the analysis is bound to one class, while the other can be fixed with some efforts. The false negative, i. e. the vulnerability the Improper state verification detector could not detect [6], will be detected if another corresponding detector is implemented to cover step 1 in the developer-oriented model [7].

8.2.2 Comparison to related work

Existing vulnerability analysis and testing frameworks for securing OpenID Connect are mostly black-box tools that can only be run in a late stage of development. The main advantage of these tools is that they are language-agnostic, and are applicable to websites that do not have public source code. They also have a complete model of the protocol flow, and get a ground-truth of sorts through the messages that are sent between the Web servers in the protocol.

Some of the black-box tools are browser extensions, which likely means that they only validate the security of a client application in a late stage when it is already in production [11, 51]. Other black-box tools analyzing protocol implementations are presented as penetration testing tools [54, 99]. These tools require manual configuration from the security tester, who has to actively herd the tool in the certain parts of the analysis, entering URLs and helping the tool visit the correct website.

The tool by Zhou et al. [99] offers some automatic passive features after the initial setup. This however introduces some challenges, as they have to impose some potentially incorrect assumptions about buttons and HTML structure of the page they analyze. For this purpose, these tools are therefore highly specified for a designated penetration tester, and are not very applicable tools for an ordinary developer.

Some of the existing automated analyses also use white-box techniques to secure OpenID connect or OAuth implementations [76, 95, 96]. Two of these use symbolic execution techniques to model the program's execution paths [95, 96]. Both tools are focused on finding vulnerabilities in SDKs, like the Nimbus SDK and Google SDK. Thus, these are not focused on what happens when a developer uses the SDK.

These also have some other problems that are solved by introducing the analysis in this thesis. Wang et al. [95] reported that their symbolic execution took between 11 and 25 hours to analyze SDKs with under 2000 lines of code. Yang et al. model their symbolic execution on paths that the attacker can see, an thereby reduce their complexity, so their analysis took no more than five seconds even for programs with 18000 lines of code.

[6]The false negative of the Improper state verification detector is described in Chapter 7.3.1.
[7]The developer-oriented model is described in Chapter 5.1.

In comparison, the static analysis in this thesis (integrated into FindSecBugs) used 2.8 seconds to analyze the largest included project, Firebase, with 7000 lines of Java code. However the analysis of SonarQube took 23 seconds since it included an unknown volume of files from the dependencies that were package into the `.jar` file that was analyzed.

The solution proposed by Rahat et al. [76], OAuthLint, is the only one of the related works that uses static analysis to detect vulnerabilities, in the OAuth protocol in Android applications. They use a formal predicate language to query a control-flow graph of the program. The vulnerabilities they target are based on the limitations of OAuth, and they have a scope that looks at different vulnerabilities related to transfer protocols and local storage of data. In comparison, the analyses in this thesis seek to cover all the authorization code flow steps in OpenID connect, with an emphasis on validation of data like the ID token and *state* parameters. Such vulnerabilities cannot be detected by OAuthLint.

PrOfESSOS [54] by Mainka and Wich is the only related token that scans applications for token forgery attacks, i.e. ID token validation vulnerabilities. They tested their penetration testing on 8 open-source Relying Party libraries. They discovered 22 ID token-related vulnerabilities which intersect those the *Improper ID token verification detector* looks for. Additionally, they found 8 vulnerabilities in a novel attack they have proposed, called *IdP Confusion*. They do not give any information about false positives, and it is therefore not easy to reason about the precision of their analysis.

Like mentioned above, their penetration testing tool only detects such vulnerabilities in already running applications, and their tool requires manual configuration. In comparison, the static analyses proposed in this thesis can mitigate these vulnerabilities during development. The *Improper ID token verification detector* found 19 equivalent vulnerabilities in six open-source code bases implementing client logic with ID tokens, plus one vulnerability they do not cover [8]. The *Improper ID token verification detector* is implemented in FindSecBugs, and can be run automatically without manual configuration.

As such, none of the other known works fills the space that this thesis proposes, as the first static analysis tool that detects ID token validation vulnerabilities. While the other tools are useful to detect vulnerabilities and protect already implemented applications, and assist penetration testers in their security analysis of an application, all these tools detect vulnerabilities very late in the development stage.

This thesis is therefore, to the best of my knowledge, the first work that uses simple static analysis techniques to detect vulnerabilities in OpenID Connect client applications. It also has the static first analysis to detect ID token verification related vulnerabilities.

While the scope of the currently implemented analyses in this thesis is not as comprehensive as several of the penetration testing tools, the main driver for this effort is to provide assistance to the developers. By implementing the analyses in a tool that is offered as an IDE plugin, vulnerabilities introduced in the code by the developers can be picked up very early in the development phase. If any vulnerabilities are impossible to find with static analysis, these may be picked up later by penetration testing tools.

[8]The unique vulnerability detected in this thesis is the warning that an incomplete SDK-implemented validator is used. This is described in Chapter 7.3.1.

For other vulnerability classes like injection, static analysis tools tend to complement penetration testing tools in what kind of vulnerabilities they find [6]. This may also be the case for certain vulnerabilities in OpenID Connect, and this work can therefore contribute well alongside penetration testing tools proposed. While the detectors implementing using the techniques presented in this thesis detect a certain set of vulnerabilities, they are limited in what information they can get about the program, and some cases are better suited for a tool used in the run-time. The same factor goes the other way, as penetration testing tools cannot reason about details in implementation errors.

8.2.3 Adding new detectors

An advantage of securing OpenID Connect applications with simple static analysis techniques is that the techniques are simple to mass-produce. To add another *Co-existing Invocation Enforcement* detector, the core algorithm will be identical. Mostly the trigger patterns and the exit patterns are what must be configured based on the data types that it is looking for. When making a new detector of this kind, there are two things to think about: 1) What is the pattern that triggers our analysis? For instance when ensuring that the token is validated, it must be a place in the code where these checks are relevant. 2) What are the correct security patterns we expect in this context to exit the analysis? Here the rules in the developer-oriented model come in. We know that if the code is requesting an ID Token, it should verify at least the five required parameters before using the token to obtain restricted user information.

Today, most of the logic in the existing detectors can be duplicated and specialized for the other cases described in Table 6.2. In a future version, the vulnerability patterns may be configured in files, allowing to easily extend the detector to cover new cases.

Such a configuration framework is implemented for the Taint analysis in FindSecBugs [70], and should be possible to add for the detectors proposed in this thesis. If the developers of the SDKs contributed to the definitions of vulnerability patterns, their insights of the data types in the libraries would prove very valuable to the prodction of precise detectors.

8.2.4 General reflections on static analysis

The research question that was to be answered, RQ2, relates to how simple static analyses can be used to cover the security of OpenID Connect. This is the prospect that really was the driving factor of the knowledge work related to RQ1, the checklist of what to do to secure the protocol, by avoiding mistakes. “Is it enough to have this simple knowledge of what is needed, by that inferring what is missing?” Then the the spirit that originally drove FindBugs [44] still stands: people make dumb mistakes. Even while implementing a modern protocol, it is still likely that developers make mistakes. These mistakes are likely to be something simple, hence many of the code defects may be easy to detect, and can be detected early.

Given the simplicity of the mistakes that then exist in a code base, then it has some logical sense that we can use simple security testing to verify that these simple things are done right. Simple static analyses have several advantages. They are often easy to use, they are easy to produce, and they can obtain a wide coverage. Through integrating the analysis with the popular FindSecBugs tool which has the FindBugs framework in its core, a large range of usability is possible. One of the main arguments for using static analysis is that bugs can be detected early in the development phase. The developer could use the tool in their IDE, or set up the analyses with their continuous integration. In both cases, vulnerabilities can be mitigated early in the development life cycle.

One inherent challenge with simple static analysis techniques is that they are prone to false positives and false negatives. They are based on a simple model with a certain set of assumptions. To some level, these assumptions must be satisfied. The analyses will be restricted by which information the detectors can retrieve from the program across the code files. If the code base is very complex and introduces many abstractions, they will no longer be covered by a model which is originally rooted in rather simple examples.

However there is an argument that security code should be as simple as possible, and that unnecessary complexity generally can make the code more prone to vulnerabilities [3]. If the code is too "dodgy" to pass through the analysis, one might argue that it is a sign of a code smell, hence, the static analysis technique may also enforce a certain coding style for the security code. Security-critical should arguably be of a higher quality, and as simple and readable as possible. Code which is readable for a human is also likely to be more readable for a static analyzer.

Therefore some false positives that come with the analysis can be considered style-warnings. So even if the true bug does not exist, the complexity of a warned code base might bring up some red lights.

8.3 Threats to validity and reliability

The threats to the internal validity of this thesis include selection bias and experimenter bias. The selection bias comes because the applications that were taken into the experiment were based on simple searches in the Github search engine. To cope with this, inclusion criteria were clearly defined before the searches were done. In addition to this, only relevant files were included from the applications, and these files had to be modified slightly, because they could not be analyzed easily out-of-the-box. This removed some realism because the entire code bases were not analyzed in their raw form. The inclusion criteria for relevant code files are explained in Chapter 7.1.

The other potential threat is experimenter bias. This bias could relate to the experimenter sub-consciously altering the methods to achieve a certain set of results. A factor here is that the selected files had to be altered to make them compile (and they had to compile to be analyzed at all). These alterations could have unknown impact on the results. This is attempted dealt with by publishing the altered code which was analyzed, and documenting all the results associated with these, as well as a clear explanation of the methods used.

The external validity of the thesis is mainly threatened by the Example-based research strategy explained in Chapter 4.3. This strategy is based on illustrating results with some real-world examples. Results from these can not be generalized due to lack of statistical significance, coming from a small data set. These code examples cannot confidently represent the average code base.

Additionally, two SDKs were selected for analysis of implementation details. These SDKs do not confidently represent all OpenID connect SDKs, and analyses related to them cannot be generalized. While it is likely that most these SDKs have many similarities due to them implementing the same protocol standard, differences can be many. For these two SDKs, special cases had to be introduced in the analyses to account for a major difference in how the same function is implemented differently.

The test-retest-reliability of the validation results is threatened by the size of the data set. If another experiment was conducted, the results risk being very different from the ones in this study. A proper large-scale empirical study is required to set a bench-mark for the ability of this tool.

This internal consistency of the results is also threatened. Between the application groups for the two SDKs, the internal correlation of subsets of the data in Table 7.6 differs greatly.

Chapter 9

Conclusions and Further Work

This thesis, has proposed a developer-oriented model of OpenID Connect. It has been implemented and employed as a foundation for static analyses designed to help developers secure the critical protocol steps. These analyses, which only cover part of the protocol, uncovered 20 vulnerabilities in six open-source applications. The research done here resulted in the first static analysis that detects ID token validation vulnerabilities. This demonstrates that simple static analyses can be used to find security bugs in OpenID connect clients.

The work in this thesis could provide both industrial and scientific value. Firstly, the analyses are implemented as extensions of the prevalent static analysis tool, Find Security Bugs. This tool is easy to use and popularly downloaded, and may enable a significant number of developers to improve the security of their OpenID Connect code. Secondly, the knowledge work done in this thesis may give further research incentive to keep putting more effort into using simple, pragmatic techniques as an avenue parallel to the complex models, for securing modern web applications.

The precision of the tool, as implemented today, was 51% in total, and 61% for the Improper ID token verification detector. Such a precision might not be satisfactory for a practitioner. However, practitioners are inclined to accept a lower precision if the code analyzed is security critical. The precision of the analyses can be increased significantly through a few weeks of careful engineering efforts. The tool also had a True Negative Rate of 89%, meaning that 9 out 10 negative cases were correctly predicted as negative, showing resistance to false positives. The recall of 95% indicates that the tool is effective in picking up vulnerabilities if they exist.

9.1 Conclusion

This thesis has attempted to answer two research questions through generating a qualitative model of OpenID Connect, and design and implementation of a software component:

RQ1 What must a developer do to avoid introducing known security vulnerabilities, while implementing a Relying Party with an OpenID Connect SDK?

RQ2 How can simple, explicit and intraprocedural static analysis checks be used to identify vulnerabilities in OpenID Connect Relying Parties?

It contributes to the body of knowledge by 1) providing a simple, more pragmatic model of OpenID Connect tailored for the implementation details of the client developer's interests, in form of a *checklist*, 2) demonstrating that simple static analysis techniques can be used to detect vulnerabilities in OpenID Connect relying parties, and 3) proposing, to the best of my knowledge, the first static analysis that detects ID token validation vulnerabilities in Relying Parties.

9.2 Further Work

9.2.1 Implementation and experimentation

For further work it would be interesting to implement the analyses through the rest of the suggested detectors in Table 6.2. Covering the whole model, and then performing a large-scale study would give statistically significant answers about the strengths and weaknesses of the analysis techniques used in this thesis, with more conclusive metrics that more confidently illustrate how the analyses perform. It would also give a better indication to what degree the model is useful for the analyses. These improvements would give generalizable results, with a more complete solution to fulfill the goal of mitigate a substantial part of the vulnerabilities that may come in OpenID Connect clients.

In the validation of the results in this study, the only detector which found vulnerabilities was the Co-existing Invocation Enforcement analysis. It is not from this known whether the other detector types are effective if true cases occur, so their potential is yet unknown. In addition, a fourth analysis may be added as a final detector type for the protocol. The data flow analysis techniques that exist in FindBugs may also be used for some appropriate cases in OpenID connect where the Static Control Flow Check comes short. This should be investigated further.

9.2.2 Reduction of false positives and negatives

The precision of tool was not initially within a range that is generally acceptable for practitioners. False positives may be reduced with a careful engineering effort of a few weeks. The analyses done in the validation trial in this thesis, have the potential to reach a precision of 83% if more work is put down to cover rare cases of complex code structure.

For the Improper ID token verification detector, 10 of its 13 false positives can be removed. These 10 false positives came due to the way the methods in the code are structured. It already has a check that infers inter-procedural checks in forwards analysis “down” the call stack. If an equivalent check is implemented for backwards calls in the call stack, this can be avoided.

The Control flow ID token verification detector had three false positives which are possible to avoid if it is extended to look for a few more patterns. To eliminate the final false positive, its algorithm have be altered to get some inter-procedural attributes similar to the ones in the Co-existing Invocation Enforcement analysis.

Lastly, for the Improper state verification detector the false negative that came can be eliminated by the introduction of another detector. Generally, more detectors will likely lead to fewer false negatives. One of its false positives can be caught if applying the same solution as for the Improper ID token verification detector. Its last false positive is not possible to pick up because of the limitations of FindBugs detectors.

9.2.3 Improvements to the developer-oriented model

Additionally, it would be interesting to refine the developer-oriented model, applying it to more of the other flows in OpenID Connect. It would also be interesting to see whether such a framework can be collated in one location for several SDKs and languages, providing specific checklists in addition to the more general specifications.

Finally, the providers of SDKs are encourage to contribute to annotating patterns for the analyses in FindSecBugs. Their knowledge about the quirks in their own code base might help increase the sophistication of the analysis. A collective effort from SDK developers would also accelerate the introduction of patterns of other SDKs that are currently not covered by the implemented detectors.

References

[1] Findbugs™ - find bugs in java programs. http://findbugs.sourceforge.net/. Accessed in April 2020.

[2] ALACA, F., AND VAN OORSCHOT, P. C. Comparative Analysis and Framework Evaluating Web Single Sign-On Systems.

[3] ALENEZI, M., AND ZAROUR, M. On the relationship between software complexity and security. *arXiv preprint arXiv:2002.07135* (2020).

[4] ALLEN, F. E. Control flow analysis. In *ACM Sigplan Notices* (1970), vol. 5, ACM, pp. 1–19.

[5] ALLEN, F. E. Interprocedural analysis and the information derived by it. In *Programming Methodology* (Berlin, Heidelberg, 1975), C. E. Hackl, Ed., Springer Berlin Heidelberg, pp. 291–321.

[6] ANTUNES, N., AND VIEIRA, M. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing* (2009), pp. 301–306.

[7] atricode-idbus: Atricore identity bus platform. https://github.com/atricore/atricore-idbus. Accessed in May 2020. Version:1.4.3-27.

[8] AYEWAH, N., PUGH, W., HOVEMEYER, D., MORGENTHALER, J. D., AND PENIX, J. Using static analysis to find bugs. *IEEE Software 25*, 5 (2008), 22–29.

[9] BENANTAR, M. *Access Control Systems: Security, Identity Management and Trust Models*. Springer, 2006.

[10] BOCIĆ, I., AND BULTAN, T. Finding access control bugs in web applications with cancheck. In *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (8 2016), Association for Computing Machinery, Inc, pp. 155–166.

[11] CALZAVARA, S., FOCARDI, R., MAFFEI, M., SCHNEIDEWIND, C., SQUARCINA, M., AND TEMPESTA, M. {WPSE}: Fortifying web protocols via browser-side security monitoring. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1493–1510.

[12] CHO, J., GARCIA-MOLINA, H., AND PAGE, L. Efficient crawling through url ordering. In *Seventh International World-Wide Web Conference (WWW 1998)* (1998).

[13] CHRISTAKIS, M., AND BIRD, C. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, ACM, pp. 332–343.

[14] Ddf distributed data framework - an open source, modular integration framework. https://github.com/codice/ddf. Accessed in May 2020. Version: ddf-2.24.0.

[15] https://connect2id.com/products/nimbus-oauth-openid-connect-sdk/examples/openid-connect. Accessed in May 2020.

[16] Idtokenvalidator. https://www.javadoc.io/doc/com.nimbusds/oauth2-oidc-sdk/latest/src-html/com/nimbusds/openid/connect/sdk/validators/IDTokenValidator.html#line.234. Accessed in May 2020, version 8.2 of the SDK.

[17] Nimbus oauth 2.0 sdk with openid connect 1.0 extensions v8.2. https://www.javadoc.io/doc/com.nimbusds/oauth2-oidc-sdk/8.2/index.html. Accessed in May 2020, version 8.2 of the SDK.

[18] Nimbus oauth 2.0 sdk with openid connect extensions. https://connect2id.com/products/nimbus-oauth-openid-connect-sdk. Accessed in May 2020.

[19] https://connect2id.com/products/nimbus-oauth-openid-connect-sdk/guides/java-cookbook-for-openid-connect-public-clients, 2018. Accessed in May 2020.

[20] https://connect2id.com/blog/how-to-validate-an-openid-connect-id-token, 2015. Accessed in May 2020.

[21] COOPER, K., AND TORCZON, L. *Engineering a compiler*. Elsevier, 2011.

[22] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. Tech. rep., 2009.

[23] Spotbugs architecture. https://github.com/spotbugs/spotbugs/blob/master/spotbugs/design/architecture/architecture.tex, 2017. Accessed in April 2020.

[24] DEEPA, G., AND THILAGAM, P. S. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology 74* (6 2016), 160–180.

[25] DEEPA, G., THILAGAM, P. S., PRASEED, A., AND PAIS, A. R. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications 109* (2018), 89–109.

[26] Delaitre, A. M., Stivalet, B. C., Black, P. E., Okun, V., Cohen, T. S., and Ribeiro, A. Sate v report: Ten years of static analysis tool expositions. Tech. rep., 2018.

[27] El Kateb, D., ElRakaiby, Y., Mouelhi, T., and Le Traon, Y. Access control enforcement testing. In *Proceedings of the 8th International Workshop on Automation of Software Test* (2013), IEEE Press, pp. 64–70.

[28] Eliassoren/find-sec-bugs. https://github.com/Eliassoren/find-sec-bugs/tree/feature/evaluation-opensource. Accessed in June 2020. Repository for the implemented detecors, in the branch used for evaluation.

[29] Oidc-findsecbugs-eval. https://github.com/Eliassoren/Oidc-FindSecbugs-Eval. Accessed in June 2020. Repository for the evalation.

[30] Emanuelsson, P., and Nilsson, U. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci. 217* (July 2008), 5–21.

[31] Espinoza, A. M., Knockel, J., Comesaña-Alfaro, P., and Crandall, J. R. V-dift: Vector-based dynamic information flow tracking with application to locating cryptographic keys for reverse engineering. In *2016 11th International Conference on Availability, Reliability and Security (ARES)* (Aug 2016), pp. 266–271.

[32] Fang, Z., Zhang, Y., Kong, Y., and Liu, Q. Static detection of logic vulnerabilities in java web applications. *Security and Communication Networks 7*, 3 (2014), 519–531.

[33] Fett, D., Küsters, R., and Schmitz, G. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the ACM Conference on Computer and Communications Security* (2016).

[34] Fett, D., Kusters, R., and Schmitz, G. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *Proceedings - IEEE Computer Security Foundations Symposium* (2017).

[35] Firebase admin java sdk. https://github.com/firebase/firebase-admin-java. Accessed in May 2020. Version: v6.13.0.

[36] Gang, F. Six ways to automatically find software bugs.

[37] https://developers.google.com/identity/protocols/oauth2/openid-connect, April 2020. Accessed in April 2020.

[38] Google identity platform. https://developers.google.com/identity, 2020. Accessed in may 2020.

[39] Google oauth client library for java 1.30.6. https://javadoc.io/doc/com.google.oauth-client/google-oauth-client/latest/overview-summary.html, 2020. Accessed in may 2020, for version 1.30.6.

[40] Idtokenverifier. https://javadoc.io/doc/com.google.oauth-client/google-oauth-client/latest/com/google/api/client/auth/openidconnect/IdTokenVerifier.html, 2020. Accessed in May 2020, for API version 1.30.6.

[41] Hardt, E. D. The oauth 2.0 authorization framework. RFC 6749, Internet Engineering Task Force (IETF), October 2012.

[42] Henry, S., and Kafura, D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering SE-7*, 5 (Sep. 1981), 510–518.

[43] Homaei, H., and Shahriari, H. R. Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour. *Information and Software Technology 107* (2019), 112–124.

[44] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (Dec. 2004), 92–106.

[45] Jones, M., M., Bradley, J., Identity, P., and Sakimura, M. Json web token (jwt). RFC 7519, Internet Engineering Task Force (IETF), May 2015.

[46] Khalid, M. N., Farooq, H., Iqbal, M., Alam, M. T., and Rasheed, K. Predicting web vulnerabilities in web applications based on machine learning. In *Intelligent Technologies and Applications* (Singapore, 2019), I. S. Bajwa, F. Kamareddine, and A. Costa, Eds., Springer Singapore, pp. 473–484.

[47] Kronjee, J., Hommersom, A., and Vranken, H. Discovering software vulnerabilities using data-flow analysis and machine learning. In *ACM International Conference Proceeding Series* (8 2018), Association for Computing Machinery.

[48] Li, J., Beba, S., and Karlsen, M. M. Evaluation of open-source ide plugins for detecting security vulnerabilities. In *Proceedings of the Evaluation and Assessment on Software Engineering* (New York, NY, USA, 2019), EASE '19, ACM, pp. 200–209.

[49] Li, W., and Mitchell, C. J. Security issues in OAUTH 2.0 SSO implementations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2014).

[50] Li, W., Mitchell, C. J., and Chen, T. Mitigating CSRF attacks on OAuth 2.0 and OpenID Connect.

[51] Li, W., Mitchell, C. J., and Chen, T. Oauthguard: Protecting user security and privacy with Oauth 2.0 and Openid connect. In *Proceedings of the ACM Conference on Computer and Communications Security* (2019).

[52] Liferay portal. https://github.com/liferay/liferay-portal. Accessed in May 2020. Version: 7.3.2 GA3.

[53] LUOTONEN, A., AND ALTIS, K. World-wide web proxies. *Computer Networks and ISDN systems 27*, 2 (1994), 147–154.

[54] MAINKA, C., MLADENOV, V., SCHWENK, J., AND WICH, T. SoK: Single Sign-On Security - An Evaluation of OpenID Connect. In *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* (2017).

[55] Facebook breach: Single sign-on of doom. https://www.bankinfosecurity.com/blogs/facebook-breach-single-sign-on-doom-p-2668.

[56] MEYER, B. Soundness and completeness: With precision. https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext, 2019.

[57] MØLLER, A., AND SCHWARZ, M. Automated Detection of Client-State Manipulation Vulnerabilities. *ACM Transactions on Software Engineering and Methodology 23*, 4 (9 2014), 1–30.

[58] MONSHIZADEH, M., NALDURG, P., AND VENKATAKRISHNAN, V. N. Mace: Detecting privilege escalation vulnerabilities in web applications. In *ACM Conference on Computer and Communications Security* (2014).

[59] MUTHUKUMARAN, D., O'KEEFFE, D., PRIEBE, C., EYERS, D., SHAND, B., AND PIETZUCH, P. Flowwatcher: Defending against data disclosure vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 603–615.

[60] Openid connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html, November 2014.

[61] Openid connect discovery 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-discovery-1_0.html, November 2014.

[62] NAVAS, J., AND BELTRÁN, M. Understanding and mitigating OpenID Connect threats. *Computers and Security* (2019).

[63] NEAR, J. P., AND JACKSON, D. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings - International Conference on Software Engineering* (5 2016), vol. 14-22-May-2016, IEEE Computer Society, pp. 947–958.

[64] NIELSON, FLEMMING;NIELSON, H. R. C. *Principles of Program Analysis.* Springer Berlin / Heidelberg, Berlin, 2015.

[65] OATES, B. J. *Researching information systems and computing.* Sage, 2005.

[66] OWASP.ORG. Category:access control. https://www.owasp.org/index.php/Category:Access_Control, 2016. Accessed in october 2019.

[67] OWASP.ORG. Fuzzing. https://www.owasp.org/index.php/Fuzzing, 2018. Accessed on 08.12.2019.

[68] PELLEGRINO, G., CATAKOGLU, O., BALZAROTTI, D., AND ROSSOW, C. Uses and abuses of server-side requests. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer, pp. 393–414.

[69] Project structure. https://github.com/find-sec-bugs/find-sec-bugs/wiki/Project-Structure, 2015. Accessed in April 2020.

[70] How "injection-based" detectors work. https://github.com/find-sec-bugs/find-sec-bugs/wiki/Injection-detection, 2017. Accessed in June 2020.

[71] Writing a detector. https://github.com/find-sec-bugs/find-sec-bugs/wiki/Writing-a-detector, 2018. Accessed in April 2020.

[72] Owasp find security bugs-the community static code analyzer. https://gosecure.github.io/presentations/2019-09-12-appsecglobaldc/OWASP_Find-Security_Bugs.pdf, 2019. Accessed in April 2020.

[73] Cli tutorial. https://github.com/find-sec-bugs/find-sec-bugs/wiki/CLI-Tutorial, March 2019. Accessed in May 2020.

[74] Find security bugs - the spotbugs plugin for security audits of java web applications. https://find-sec-bugs.github.io/, 2019. Accessed in April 2020.

[75] POTTER, B., AND MCGRAW, G. Software security testing. *IEEE Security Privacy 2*, 5 (Sep. 2004), 81–85.

[76] RAHAT, T. A., FENG, Y., AND TIAN, Y. Oauthlint: An empirical study on oauth bugs in android applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (2019), ASE '19, IEEE Press, p. 293–304.

[77] RAZZAQ, A., LATIF, K., FAROOQ AHMAD, H., HUR, A., ANWAR, Z., AND BLOODSWORTH, P. C. Semantic security against web application attacks. *Information Sciences 254* (1 2014), 19–38.

[78] ROCHE, E., AND SCHABES, Y. *Finite-state language processing.* MIT press, 1997.

[79] RUSSELL, R., KIM, L., HAMILTON, L., LAZOVICH, T., HARER, J., OZDEMIR, O., ELLINGWOOD, P., AND MCCONLEY, M. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018), IEEE, pp. 757–762.

[80] SADQI, Y., BELFAIK, Y., AND SAFI, S. Web oauth-based sso systems security. In *Proceedings of the 3rd International Conference on Networking, Information Systems Security* (New York, NY, USA, 2020), NISS2020, Association for Computing Machinery.

[81] SENG, L. K., ITHNIN, N., AND SAID, S. Z. M. The approaches to quantify web application security scanners quality: a review. *International Journal of Advanced Computer Research 8*, 38 (2018), 285–312.

[82] SHAW, M. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer 4*, 1 (2002), 1–7.

[83] SON, S., AND SHMATIKOV, V. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security* (2011), ACM, p. 8.

[84] https://github.com/spotbugs/spotbugs, 2019. Accessed in April 2020.

[85] SUN, F., XU, L., AND SU, Z. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium* (2011), vol. 64.

[86] SUN, S. T., AND BEZNOSOV, K. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012).

[87] SØRENSEN, E. B. A literature review and practitioner survey on using vulnerability detection tools to defend against access control vulnerabilities. https://www.researchgate.net/publication/338396329_A_Literature_Review_and_Practitioner_Survey_on_Using_Vulnerability_Detection_Tools_to_Defend_Against_Access_Control_Vulnerabilities, 12 2019.

[88] SØRENSEN, E. B., KARLSEN, E. K., AND LI, J. What norwegian developers want and need from security-directed program analysis tools: A survey. In *Proceedings of the Evaluation and Assessment in Software Engineering* (New York, NY, USA, 2020), EASE '20, Association for Computing Machinery, p. 505–511.

[89] T. LODDERSTEDT, ED., M. MCGLOIN, P. H. OAuth 2.0 Threat Model and Security Considerations. Tech. rep., 2013.

[90] THARWAT, A. Classification assessment methods. *Applied Computing and Informatics* (2018).

[91] https://cwe.mitre.org/data/definitions/1200.html, 2019. Added in November 2019.

[92] Owasp top 10 - 2017 the ten most critical web application security risks. https://www.owasp.org/images/b/b0/OWASP_Top_10_2017_RC2_Final.pdf, 2017. Added in September 2019.

[93] Spotbugs - find bugs in java programs. https://spotbugs.github.io/, 2019. Accessed in April 2020.

[94] Openid connect (oidc) plugin for sonarqube. https://github.com/vaulttec/sonar-auth-oidc. Accessed in May 2020. Version: v2.0.0.

[95] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the 22nd USENIX Security Symposium* (2013).

[96] Yang, R., Lau, W. C., Chen, J., and Zhang, K. Vetting single sign-on {SDK} implementations via symbolic reasoning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1459–1474.

[97] Yang, R., Li, G., Lau, W. C., Zhang, K., and Hu, P. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *ASIA CCS 2016 - Proceedings of the 11th ACM Asia Conference on Computer and Communications Security* (may 2016), Association for Computing Machinery, Inc, pp. 651–662.

[98] Zheng, Y., and Zhang, X. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 652–661.

[99] Zhou, Y., and Evans, D. SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[100] Zhu, J., Chu, B., and Lipford, H. Detecting privilege escalation attacks through Instrumenting web application source code. In *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT* (6 2016), vol. 06-08-June-2016, Association for Computing Machinery, pp. 73–80.

[101] Zhu, J., Chu, B., Lipford, H., and Thomas, T. Mitigating access control vulnerabilities through interactive static analysis. In *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT* (6 2015), vol. 2015-June, Association for Computing Machinery, pp. 199–209.

[102] Zop app. https://github.com/zopspace/zop-app. Accessed in May 2020. Version: v0.1.

Appendix A

Research paper of precursory work

What Norwegian Developers Want and Need From Security-Directed Program Analysis Tools: A Survey

Elias Brattli Sørensen
Norwegian University of Science and Technology
eliasbs@alumni.ntnu.no

Edvard Kristoffer Karlsen
Kantega AS
edvard.karlsen@kantega.no

Jingyue Li
Norwegian University of Science and Technology
jingyue.li@ntnu.no

ABSTRACT

Code enforcing access control policies often has high inherent complexity, making it challenging to test using only classical review and testing techniques. To more thoroughly test such code, it is strategic to also use program analysis tools, which often can find subtle, critical bugs going unnoticed to humans. These powerful tools are however rarely used in software consultancy practice, due to factors such as bad usability or unsatisfactory non-functional characteristics. To encourage wider adoption of such tools, more must be learned about how to design them to the preferences of software consultants. Towards this goal, we conducted a survey of Norwegian software consultants. Among our findings is a positive relation between preference for soundness over completeness in tools and preference for annotation-based over automated tools. 51% of the developers surveyed prefer soundness over completeness when detecting access control vulnerabilities, while only 37.5% view completeness as the more important characteristic. Qualitative responses illuminate concerns regarding usability, soundness, completeness, and performance.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; *Access control*; • **Software and its engineering → Software testing and debugging**; *Formal software verification.*

KEYWORDS

program analysis, static analysis, survey, access control vulnerabilities, consultants

ACM Reference Format:
Elias Brattli Sørensen, Edvard Kristoffer Karlsen, and Jingyue Li. 2020. What Norwegian Developers Want and Need From Security-Directed Program Analysis Tools: A Survey . In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3383219.3383293

1 INTRODUCTION

Access control vulnerabilities (ACVs) regularly result in critical data leaks in software systems. When the application Æ, used by customers of the Norwegian supermarket chain REMA 1000, was launched, its back-end API allowed unauthorized retrieval of sensitive data [7]. Another example is the Facebook scandal in April 2019 [3], where user data for hundreds of millions of users was exposed on a cloud server. ACVs often occur when code implementing authentication checks or enforcing authorization rules has faulty logic. Such subtle errors can often be found by program analysis tools, which model and reason about the behavior of programs. However, these tools are regrettably seldom used by software consultants, due to factors like bad usability or unsatisfactory non-functional characteristics[1] [14, 29].

To encourage wider adoption of program analysis tools, more must be learned about developers' requirements to these assets. This paper presents results from a survey of Norwegian software consultants, aiming to investigate the following research questions:

RQ1 What non-functional requirements do consultants have for program analysis tools for detecting ACVs?

RQ2 How does the background of consultants affect their relative preferences for the opposing tool characteristics *soundness versus completeness* and *automatic versus annotation-based*?

Quantitative and qualitative data support the following principal findings for **RQ1**:

- High soundness is considered more important than high completeness when uncovering ACVs.
- There is a near 50/50 preference distribution between fully automated and annotation-based tools.
- Seamless workflow integration may increase the chance program analysis tools are used.
- Most of the respondents reply false positive rates should not exceed 10%.

Regarding **RQ2**, hypothesis tests show that neither degree of general experience, experience with security-critical system, nor amount of security-oriented education significantly influence developers' relative preferences for the opposing tool characteristics soundness versus completeness and automatic versus annotation-based.

In addition, we identify a weak, positive relation between preference for soundness over completeness in tools and preference for annotation-based over automated tools.

The remainder of the text is structured as follows: Section 2 presents related work. Section 3 describes the research design. Section 4 presents quantitative results for RQ1 and RQ2, while Section 5 presents qualitative results. Section 6 discusses the results and threats to validity. Finally, Section 7 concludes and suggests ideas for further work.

[1] For purpose of this study, we consider the important non-functional characteristics workflow integration point, computational efficiency, and soundness and completeness.

2 RELATED WORK

Christakis and Bird conducted a survey targeting developers in Microsoft [4]. They investigated developers' requirements to static analyzers and compiled a ranked list of barriers against their use. Among their primary findings were:

- Developers want the opportunity to customize analyzers.
- Programmatic annotations are most preferable, before rules given in global configuration files and annotations coded in comments.
- 90% of developers accept 5% false positives, 50% of developers accept 15% false positives, and only 24% of developers accept more than 20% false positives.
- There is a 50/50 distribution in preferences for soundness versus completeness.

Thomas et al. [27] conducted a study investigating the implications of interactive code annotations within the IDE. Their main findings were that it is easy to write annotations for access control logic, but hard to find causes of vulnerabilities. Even non-security people were able to describe access control policies with reasonable effort.

Sadowski et al. [22] worked with a Google project called Tricorder. Their main findings include:

- Low false alarm rate is important.
- It is important to allow customization at project level, and not only at user level.
- Analysis tools should not only find, but also fix bugs. Tools that automatically apply fixes reduce the need for context switches.
- Program analysis tools should be shardable to ensure that analyses can run at large scale.

Tripp et al. [28] present a tool called ALETHEIA. Their main idea is to apply statistical learning to user-tailor warning output. The tool learns from feedback on a smaller set of warnings. They confirm the well-known finding that developers are very bothered by an excess of false positives.

Tymchuk et al. [29] interviewed experienced developers to understand how they were influenced by an IDE tool providing just-in-time feedback for good coding practices. Usefulness of analyzers in different situations was assessed, and they gathered feedback about the behavior of the tool. They found that the main negative issues of static analyzers are false positives, unclear explanations, annoying user experience, and annoying rules.

Li et al. [14] performed an experimental validation of various open-source IDE plugins that detect security vulnerabilities. They investigated vulnerability class scopes, quality of detection, and user-friendliness of tool warnings. They found a mismatch between the claimed and actual coverage of the tools, as well as unexpectedly high false positive rates. Several tools had limited information in their output, with drawbacks such as imprecise or lacking explanations of vulnerabilities. Another issue was missing opportunities to direct a tool; some tools are only able to scan full code bases, and not smaller units.

3 RESEARCH DESIGN

To direct the design of the study, we conducted a review [26] of relevant research in program analysis tools for detecting access control vulnerabilities [2, 5, 6, 8–11, 13, 15–18, 20, 21, 24, 25, 31–33]. Research guidelines suggested by Kitchenam and Pfleeger [12] and by Oates [19] were also considered in the design process. Christakis and Bird's survey of Microsoft developers [4] was a particularly important influence.

The main purpose of the questionnaire was to explore developers' relative preferences between opposing tool characteristics, and their thoughts on various challenges with and requirements of tools.

The questionnaire was distributed to approximately 750 consultants, from seven consultancy firms. Each of the invitees received a reminder a few days after the initial invitation, and the questionnaire was open for a week.

After reviewing relevant literature and similar surveys, six statistical hypotheses, listed in Table 1, were selected. While hypotheses 1–5 lay wholly within the scope of RQ2, hypothesis 6, which does not concern a background-specific relation, does not.

Two details concerning the formulation of the six hypotheses should be clarified: First, many hypotheses, and survey questions, concern the *relative preference* between soundness and completeness, on the underlying assumption that increased performance with regard to one attribute necessitates a decrease in performance with the other attribute . Thus, when the phrase "prefers soundness over completeness" is used, it means only that one is willing to sacrifice some degree of completeness for increased soundness, not that one would not ideally want both. Second, for brevity, the qualifier "software consultants" is generally omitted from the hypotheses, and the shorthand "tools" is used to mean specifically "program analysis tools for detecting access control vulnerabilities".

Inferential statistical analysis involving ordinal data is a contested, methodologically challenging issue [1, 23, 30]. It is especially difficult to assess when classical parametric statistical tests are applicable, and how to safely prepare ordinal data for use with such tests. To err on the side of caution, we opted to use only non-parametric tests in the analysis. In particular, Kendall's Tau rank correlation coefficient was used for the majority of hypotheses, as it is a natural choice for investigating relationships involving ordinal variables representing preferences. The downside of using non-parametric tests is that they generally have lower statistical power than parametric ones; non-parametric tests are more likely to result in type II errors, where one fails to reject a false null hypothesis. For purpose of our study, we regard it preferable to err on the side of rejecting a hypothesized relation, rather than erroneously concluding one exists when that is not the case.

3.1 Semi-structured interviews

Three respondents were invited to semi-structured, follow-up interviews after participating in the questionnaire. The focus of these interviews was to get insight into how these respondents interpreted the questions and gave their responses, to discover potential weaknesses in the survey design and enhance understanding of the survey data and results. The qualitative responses were analyzed semantically, though not with any formal coding framework.

4 RESULTS

Of the approximately 750 consultants invited to participate in the questionnaire, 80 persons responded. Among these, 87% primarily write and maintain (production) code, 5% work as testers, 4%

Table 1: Statistical Hypotheses

Null hypotheses	**Alternative hypotheses**
H1$_0$ Consultants working with security-critical systems do not tend to have higher relative preference for annotation-based versus automated tools than consultants who do not work with such systems.	**H1** Consultants working with security-critical systems tend to have higher relative preference for annotation-based versus automated tools than consultants who do not work with such systems.
H2$_0$ There is not a positive relation between level of experience and preference for completeness over soundness in tools.	**H2** There is a positive relation between level of experience and preference for completeness over soundness in tools.
H3$_0$ There is not a positive relation between level of experience working with security-critical systems and preference for soundness over completeness in tools.	**H3** There is a positive relation between level of experience working with security-critical systems and preference for soundness over completeness in tools.
H4$_0$ There is not a positive relation between amount of security-oriented education and preference for soundness over completeness in tools.	**H4** There is a positive relation between amount of security-oriented education and preference for soundness over completeness in tools.
H5$_0$ There is not a positive relation between amount of security-focused education and preference for annotation-based over automated tools.	**H5** There is a positive relation between amount of security-focused education and preference for annotation-based over automated tools.
H6$_0$ There is not a positive relation between preference for soundness over completeness in tools and preference for annotation-based over automated tools.	**H6** There is a positive relation between preference for soundness over completeness in tools and preference for annotation-based over automated tools.

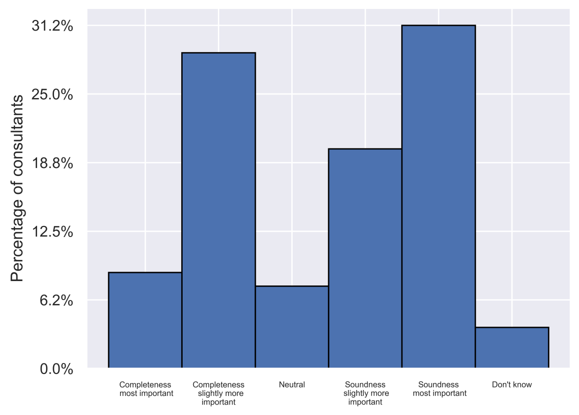


Figure 1: Distribution of relative preference for soundness versus completeness

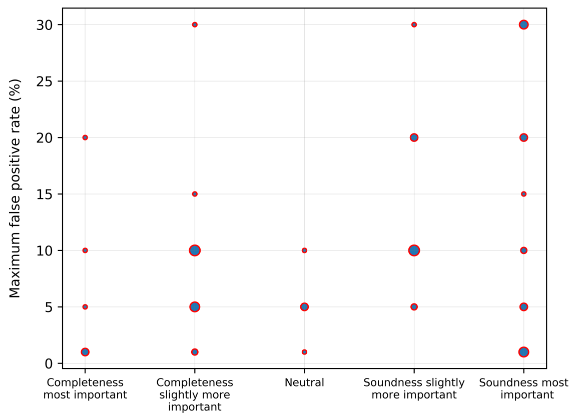


Figure 2: Joint distribution of relative preference for soundness vs. completeness and accepted rate of false positives

work as system administrators, and the remaining 4% work with management.

4.1 RQ1: Non-functional characteristics

The survey explored where in the development cycle consultants would prefer to use a program analysis tool, i.e. their preferred workflow integration point. "Direct integration in an IDE" was the most popular option, before "integration in a Continuous Integration/Continuous Delivery (CI/CD) pipeline" and "integration in the (local) build process". Further, the consultants were asked to indicate their relative preference between the conflicting attributes soundness and completeness. The ordinal data illustrated in Figure 1 show a total of 51% preferring to find as many critical errors as possible (soundness), while only 37.5% of the consultants viewed having fewer false positives as the more important attribute when detecting data leaks.

Figure 2 displays the relation between the answers for relative preference for soundness vs. completeness and accepted rate of false positives.

The consultants were also asked to weigh the characteristics "more automatic, but less precise" and "more precise, but more work with annotations" against each other. The bar plot in Figure 3 shows that the majority of the respondents lean towards the neutral ground, which highlights the importance of balance in the tools.

4.2 RQ2: Background-related effects

Hypothesis 1: *Consultants working with security-critical systems tend to have higher relative preference for annotation-based versus automated tools than consultants who do not work with such systems.*

Among the respondents, 49 persons work with security-critical systems, while 19 persons do not work with such systems. To test

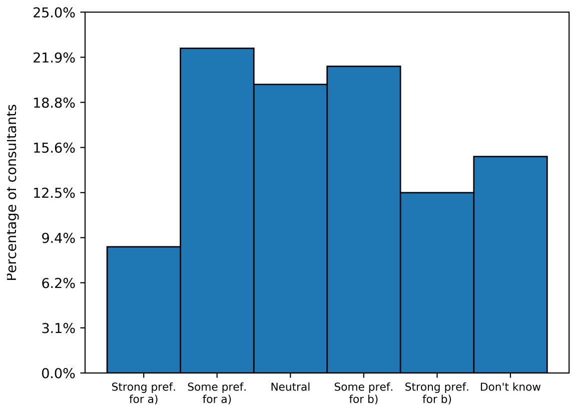


Figure 3: Distribution of relative preference between the characteristics a) "more automatic, but less precise" and b) "more precise, but more work with annotations"

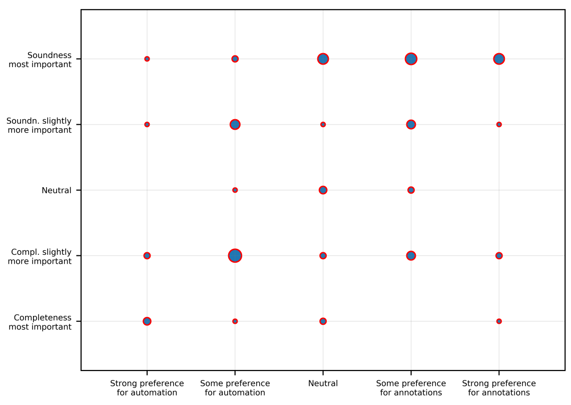


Figure 4: Joint distribution of relative preferences for soundness vs. completeness and for automatic vs. annotation-based tools

the hypothesis, the Mann–Whitney U statistic was calculated, with input the corresponding data sets representing respondents' relative preferences for automatic versus annotation-based tools. The computed U value was 522.5, and the corresponding p-value 0.21. Hence, we cannot reject the null hypothesis.

Hypothesis 2: *There is a positive relation between level of experience and preference for completeness over soundness in tools.*

Hypothesis 2 concerns the relation between a scalar, the length of a respondent's career, expressed in years, and an ordinal value indicating relative preference between soundness and completeness. To test the hypothesis, Kendall's Tau-b rank correlation coefficient was calculated. The statistic τ was 0.12, and the corresponding p-value 0.22. Thus, we cannot reject the null hypothesis.

Hypothesis 3: *There is a positive relation between level of experience working with security-critical systems and preference for soundness over completeness in tools.*

Hypothesis 3 concerns the relation between a ordinal value representing a consultant's length of experience working with security-critical systems, and an ordinal value indicating relative preference between soundness and completeness. To test the hypothesis, Kendall's Tau-b rank correlation coefficient was calculated. The statistic τ was -0.14, and the corresponding p-value 0.16. Thus, we cannot reject the null hypothesis.

Hypothesis 4: *There is a positive relation between amount of security-oriented education and preference for soundness over completeness in tools.*

Hypothesis 4 concerns the relation between an ordinal value representing a consultant's level of security-focused education and an ordinal value indicating relative preference between soundness and completeness. Kendall's Tau-b rank correlation coefficient was calculated. The statistic τ was -0.055, and the corresponding p-value 0.57. Thus, we cannot reject the null hypothesis.

Hypothesis 5: *There is a positive relation between amount of security-focused education and preference for annotation-based over automated tools.*

Hypothesis 5 concerns the relation between an ordinal value indicating a consultant's level of security-focused education and an ordinal value indicating relative preference between fully-automatic and annotation-based tools. Kendall's Tau-b rank correlation coefficient was calculated. The statistic τ was 0.16, and the corresponding p-value 0.12. Thus, we cannot reject the null hypothesis.

Hypothesis 6: *There is a positive relation between preference for soundness over completeness in tools and preference for annotation-based over automated tools.*

Hypothesis 6 concerns the relation between an ordinal value indicating relative preference between soundness and completeness and an ordinal value indicating relative preference between automatic and annotation-based tools. Kendall's Tau-b rank correlation coefficient was calculated. The statistic τ was 0.31, and the corresponding p-value 0.002. Hence, the alternative hypothesis was accepted. The data suggests a weak, but statistically significant relation between the variables ($\alpha < 0.01$).

Figure 4 shows a scatter plot created from the responses of the 66 persons that responded to both questions considered for this hypothesis.

5 QUALITATIVE RESPONSES

To get further insights for answering RQ1, the questionnaire contained open-ended questions asking the consultants for their thoughts on where to fit a program analysis tool into their workflow, how they would like vulnerabilities reported, which other vulnerabilities they find important, and what challenges they see with using program analysis tools for security. The follow-up interviews also gave valuable insights.

5.1 Workflow integration point

After giving their opinion on their preferred workflow integration point, each participant was asked to elaborate further. Several interesting responses provided potentially valuable insights. One consultant discussed possibilities of integrating program analysis in the implementation step in the development cycle: *"In the day to day basis I would find it natural that program analysis is executed in the build process, for example in Jenkins".*

An experienced consultant provided other perspectives in their written response: *"I think this is difficult to answer. For me it is natural that such access control is something that is tested in integration test and is defined in code. It should be part of an active development of an API, and one should construct it 100% restrictively, to then open the security following needs. For me this is a part of the craft of doing software development. If we should have some architecture that 'automagically' understands business rules, then it is nice to have it running in the production systems, either as firewall or other rule-based systems. But the rules must still be written?"*

The last response indicates a natural skepticism, and points to the complexity of most software systems, which makes it hard to trust that a tool can handle such complexity. The response also suggests a lower need for pedagogical tooling for more senior developers. Several consultants liked the idea of using program analysis tools during code review, or as part of a CI/CD pipeline.

5.2 Vulnerability report formats

After rating various vulnerability report methods and formats, respondents were asked for their own suggestions. One developer would like the opportunity to have access control vulnerabilities trigger compilation errors, so the code can not execute until the issue is fixed. Several respondents pointed out that they would like the output of program analysis in logs. That way they may configure dashboard-based, mail-based or other types of reporting on their own.

An important aspect of having the tool output during build or in CI/CD, was that the build or pipeline must break if there is a vulnerability. Otherwise the vulnerability report could easily drown among other log warnings. One respondent wanted the output to result in warnings in the local build process, but to result in errors when code is processed in a CI pipeline. It was pointed out how program analysis could, and should, be used in harmony with other protection methods: *"Whatever can be detected automatically should be detected as early as possible, then via either IDE, build or CI. Meanwhile, I would believe that some things are detectable only via a larger penetration test that is carried out by experts, who then typically would write a report from their test."*

Others worried about the time load tools could carry with them: *"One would not like things to take a long time, so the IDE is preferable. However, not if it heavily burdens the performance of the IDE, then it is better to put analysis later. So the answer to the questions depends on how high a load the tool puts on each step."* This respondent also highlighted the importance of analyzing during run-time in addition to static analysis. This motivation for several analysis modes may come from that attacks vectors change over time.

5.3 Challenges with program analysis

In the end of the questionnaire the respondents were asked to describe any challenges they could see with using program analysis to detect access control vulnerabilities.

The following are some of the challenges with program analysis tools that the developers in consulting mentioned:

- Properly detecting complex patterns and contexts is challenging.
- Developers may turn off an analysis if it takes too long.
- Results of analysis may appear in a hidden place, somewhere the developer must actively seek.
- The tool may be too generic for the domain.
- Result credibility is weakened with too many false alarms.
- There is lacking trust that a tool will be able to detect errors, due to the complexity in software development
- Poor performance in soundness or completeness is challenging.
- There are probably situations in which non-standard program behavior is misunderstood by the tool.
- When the tool usability is too bad developers will not use it.

5.4 Follow-up interviews

Three of the consultants were taken in to follow-up interviews, in which they got to review their questionnaire responses and provide thoughts about the topics in question.

The following overall insights and opinions came from these three interviews:

- High soundness and completeness is more important than where in the workflow a tool is used.
- A tool that learns from code practices and version control history may be valuable.
- The idea that the tool has a faster in-editor mode, and a slower mode that runs later is acceptable.
- The false positive rate may be higher if vulnerabilities are presented in an orderly, ranked manner.
- License fees are a possible barrier from usage of program analysis tools.
- A configurable tool sounds intriguing. However, the configuration must be easily understandable, and the defaults must be sensible.
- One of the respondents thinks a tool should focus on finding and ranking more intricate vulnerabilities.

6 DISCUSSION

6.1 Comparison with related work

The data illustrated in Figure 1 shows that nearly 1.4 times as many chose soundness, suggesting significant difference. 31% of the respondents found soundness to be "most important" (a score of 5). Including the scores of "most important" and "slightly more important" as weights, would give a preference ratio of nearly 1.5 in the favor of soundness, which further solidifies the overall preference. This study uses a narrower range for false positive acceptance rates than what was used by Christakis and Bird [4], illustrated in Figure 2. There is a misalignment between what is considered few false positives by participants, and what is considered few in research [4]. This apparent cognitive dissonance may explain the contrast that comes from the majority also viewing soundness as the most important factor. Interestingly, as indicated by the follow-up interviews, developers may prefer completeness in the early stage of development, and allow soundness in the later stages together with other thorough testing. The most preferred workflow integration point, embedding inside an IDE, aligns with the findings of Christakis and Bird. However as indicated by interviewees, the preference workflow integration as well as other responses depend on where in the development process the project is. The preference for using annotations had a near 50/50 distribution, as shown in Figure 3. An IDE-integrated tool should be fast, while a CI-based tool could possibly be allowed to scan code all night. Worry about license fees is one barrier against adapting tools in consulting, as well as lack of trust in the performance of open-source tools. These participants' worries are also confirmed by recent research [14].

Neither degree of general experience, experience with security-critical system, nor amount of security-oriented education significantly influence the relative preferences for the opposing tool characteristics soundness versus completeness and automatic versus annotation-based of developers.

The results of the hypothesis tests do not suggest any obvious, new guidelines for tailoring an analysis tool to the preferences of background-specific subsets of developers. However, hypothesis test 6 suggests that there exists a subset of developers who are positively inclined towards tools and more willing to make sacrifices to utilize their strengths in the development process. Still, most developers will avoid using a tool that has bad usability or is lacking in non-functional characteristics. Therefore, designers of program analysis tools should adapt to the process of software developers in order to provide proper value to the development, a point that confirms ideas from related works [4, 22, 28, 29].

Several solutions in the state of the art of program analysis for access control analysis do not have a clear usability perspective. Even the ones claiming to use "interactive communication" as a usability factor in their solution [33], have been found to come with major drawbacks regarding usability and other non-functional characteristics [14]. The worry about false positives is ever apparent, and it is unclear what should be a realistic false positive rate, though the preference towards soundness when mitigating data leaks suggests some acceptance. The developers do not want a strictly automatic or strictly annotation-based tool, but prefer to use something adaptable.

6.2 Threats to validity

The internal validity of the practitioner survey is threatened by biased and imprecise questions that still persisted after trials of testing. Another internal limitation is different understanding of terms. A term properly defined before the question may be missed or skipped by the respondent. Qualitative responses were translated from Norwegian, which carries the risks of semantics getting lost in translation.

The external validity is mainly threatened by sampling bias. A few consulting firms that were accessible through contacts of researchers were selected, and among them most consultants were invited. No probabilistic sampling was done, and the survey relies on self-selection. The mass of the respondents may still be large enough so that results can generalize to other consulting firms in Norway. The questionnaire was sent out to around 750 consultants, among which 400 were invited by e-mail, while the remaining 350 were invited by channel posts in work place chat services. Use of chat service rather than email imposes a greater risk of several potential participants never being properly exposed to the invitation, as the message quickly drowns. The response rate of 10.5% may threaten the generalizability of the study, but given that the 80 respondents come from seven different firms with various business areas, the sample may be an acceptable representation of the Norwegian IT consulting industry. The validity of comparison to related work is also threatened by differences in development culture, so it is hard to draw conclusions reaching outside of Norway for this sample. Additionally, each industry may have different software development life cycles, standards and environments, which means that the preferences of these consultants may not apply for developers with slight differences regarding these factors.

7 CONCLUSION AND FURTHER WORK

This paper surveys and analyses the preferences of Norwegian software consultants in program analysis tools for detecting access control vulnerabilities. 80 IT consultants from seven Norwegian consulting firms were surveyed for their opinions, with embedded long text answers and follow-up interviews.

We find that high soundness is considered more important than high completeness when uncovering ACVs, and observe a near 50/50 preference distribution between fully automated and annotation-based tools. Of the developers surveyed, 51% prefer soundness over completeness when detecting ACVs, and only 37.5% consider completeness the more important characteristic.

The quantitative analysis shows that neither degree of general experience, experience with security-critical system, nor amount of security-oriented education significantly influence developers' relative preferences for the opposing tool characteristics *soundness versus completeness* and *automatic versus annotation-based*.

However, the survey data suggests there exists a group of developers who are more positively inclined towards program analysis tools and more willing to make sacrifices to utilize their strengths in the development process.

The preferences regarding opposing characteristics explored in this paper may be determined by additional context-dependent factors, like project life cycle stage and kind of vulnerability. Hence, an interesting avenue for future work is to delve deeper into the various contexts to explore subtle influences over preference of tool usage. There may also exist other relations like the one explored by hypothesis 6, which could be explored. Finally, it would be interesting to look deeper into the statistical nature of the opinions in a larger-scale study with probabilistic sampling, potentially expanding to a wider population.

REFERENCES

[1] Phillip A Bishop and Robert L Herron. 2015. Use and misuse of the likert item responses and other ordinal measures. *International journal of exercise science* 8, 3 (2015), 297.

[2] Ivan Bocić and Tevfik Bultan. 2016. Finding access control bugs in web applications with cancheck. In *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, 155–166. https://doi.org/10.1145/2970276.2970350

[3] CBC News 2019. Hundreds of millions of Facebook user records were exposed on Amazon cloud server. https://www.cbsnews.com/news/millions-facebook-user-records-exposed-amazon-cloud-server/. Accessed on 19.12.2019.

[4] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[5] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. *Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications.* Technical Report. http://www.usenix.org/events/sec09/tech/full_papers/dalton.pdfhttp://hdl.handle.net/1721.1/62182http://creativecommons.org/licenses/by-nc-sa/3.0/

[6] G Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. 2018. DetLogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications* 109 (2018), 89–109.

[7] digi.no [n.d.]. Kundedata lå åpent tilgjengelig i Rema 1000s Æ-app. – Jeg kunne lastet ned hele kundebasen. https://www.digi.no/artikler/kundedata-la-apent-tilgjengelig-i-rema-1000s-ae-app-jeg-kunne-lastet-ned-hele-kundebasen/375923. Accessed on 19.12.2019.

[8] Donia El Kateb, Yehia ElRakaiby, Tejeddine Mouelhi, and Yves Le Traon. 2013. Access control enforcement testing. In *Proceedings of the 8th International Workshop on Automation of Software Test*. IEEE Press, 64–70.

[9] Zhejun Fang, Yuqing Zhang, Ying Kong, and Qixu Liu. 2014. Static detection of logic vulnerabilities in Java web applications. *Security and Communication Networks* 7, 3 (2014), 519–531.

[10] Hossein Homaei and Hamid Reza Shahriari. 2019. Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour. *Information and Software Technology* 107 (2019), 112–124.

[11] Muhammad Noman Khalid, Humera Farooq, Muhammad Iqbal, Muhammad Talha Alam, and Kamran Rasheed. 2019. Predicting Web Vulnerabilities in Web Applications Based on Machine Learning. In *Intelligent Technologies and Applications*, Imran Sarwar Bajwa, Fairouz Kamareddine, and Anna Costa (Eds.). Springer Singapore, Singapore, 473–484.

[12] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. *Personal Opinion Surveys.* Springer London, London, 63–92. https://doi.org/10.1007/978-1-84800-044-5_3

[13] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering software vulnerabilities using data-flow analysis and machine learning. In *ACM International Conference Proceeding Series*. Association for Computing Machinery. https://doi.org/10.1145/3230833.3230856

[14] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities. In *Proceedings of the Evaluation and Assessment on Software Engineering (EASE '19)*. ACM, New York, NY, USA, 200–209. https://doi.org/10.1145/3319008.3319011

[15] Anders Møller and Mathias Schwarz. 2014. Automated Detection of Client-State Manipulation Vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 23, 4 (9 2014), 1–30. https://doi.org/10.1145/2531921

[16] Maliheh Monshizadeh, Prasad Naldurg, and V. N. Venkatakrishnan. 2014. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *ACM Conference on Computer and Communications Security.*

[17] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. 2015. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 603–615.

[18] Joseph P. Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings - International Conference on Software Engineering*, Vol. 14-22-May-2016. IEEE Computer Society, 947–958. https://doi.org/10.1145/2884781.2884836

[19] Briony J Oates. 2005. *Researching information systems and computing.* Sage.

[20] Abdul Razzaq, Khalid Latif, H. Farooq Ahmad, Ali Hur, Zahid Anwar, and Peter Charles Bloodsworth. 2014. Semantic security against web application attacks. *Information Sciences* 254 (1 2014), 19–38. https://doi.org/10.1016/j.ins.2013.08.007

[21] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.

[22] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 598–608.

[23] Roger N Shepard. 1966. Metric structures in ordinal data. *Journal of Mathematical Psychology* 3, 2 (1966), 287–315.

[24] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security.* ACM, 8.

[25] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications.. In *USENIX Security Symposium*, Vol. 64.

[26] Elias Brattli Sørensen. 2019. A Literature Review and Practitioner Survey on Using Vulnerability Detection Tools to Defend Against Access Control Vulnerabilities. https://www.researchgate.net/publication/338396329_A_Literature_Review_and_Practitioner_Survey_on_Using_Vulnerability_Detection_Tools_to_Defend_Against_Access_Control_Vulnerabilities. https://doi.org/10.13140/RG.2.2.19118.05443

[27] Tyler Thomas, Bill Chu, Heather Lipford, Justin Smith, and Emerson Murphy-Hill. 2015. A study of interactive code annotation for access control vulnerabilities. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 73–77.

[28] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 762–774.

[29] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT feedback: what experienced developers like about static analysis. In *Proceedings of the 26th Conference on Program Comprehension.* ACM, 64–73.

[30] Paul F Velleman and Leland Wilkinson. 1993. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician* 47, 1 (1993), 65–72.

[31] Yunhui Zheng and Xiangyu Zhang. 2013. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 652–661.

[32] Jun Zhu, Bill Chu, and Heather Lipford. 2016. Detecting privilege escalation attacks through Instrumenting web application source code. In *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, Vol. 06-08-June-2016. Association for Computing Machinery, 73–80. https://doi.org/10.1145/2914642.2914661

[33] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. 2015. Mitigating access control vulnerabilities through interactive static analysis. In *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, Vol. 2015-June. Association for Computing Machinery, 199–209. https://doi.org/10.1145/2752952.2752976

Appendix B

Raw data from evaluation

For five of the applications, files analyzed were retrieved from the original sources [7, 14, 35, 52, 102], and had to be slightly altered so that they could be analyzed. The altered code can be found in the Eval project [29]. The sixth application, SonarQube [94], was build and analyzed with its original code.

B.1 Raw data evaluation: ZopSpace [102]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/Oidc-FindSecbugs-Eval/Eval/zopspace/target/zopspace-1.0-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	1	0.00
Medium Priority Warnings	5	0.00
Total Warnings	**6**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

Summary

Warning Type	Number
Security Warnings	6
Total	**6**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code	Warning
SECIIDTV	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo1. Bug type INCOMPLETE_ID_TOKEN_VERIFICATION (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.isValidIdToken(String, String) At OIDCUtils.java:[lines 153-162]
SECMVIDT	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo2. Bug type MISSING_VERIFY_ID_TOKEN (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.refreshTokens(String, String, String, String[], String) At OIDCUtils.java:[lines 133-145]
SECMVNONCE	ID Tokens must be validated by the Relying Party (Client). The nonce is a cryptographically opaque value like the state value which binds an authentiaction request to the ID Token Bug type MISSING_VERIFY_NONCE (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.isValidIdToken(String, String) At OIDCUtils.java:[lines 153-162]
SECMVTISSU	ID Tokens must be validated by the Relying Party (Client). iss parameter validation Bug type MISSING_VERIFY_TOKEN_ISS (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.isValidIdToken(String, String) At OIDCUtils.java:[lines 153-162]
SECMVTSIGN	ID Tokens must be validated by the Relying Party (Client). Cryptographic validation Bug type MISSING_VERIFY_TOKEN_SIGN (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.isValidIdToken(String, String) At OIDCUtils.java:[lines 153-162]
SECUIDTV	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified. Bug type USING_INCOMPLETE_ID_TOKEN_VALIDATOR (click for details) In class oidc.OIDCUtils In method oidc.OIDCUtils.isValidIdToken(String, String) At OIDCUtils.java:[lines 153-162]

Details

INCOMPLETE_ID_TOKEN_VERIFICATION: Missing one or more of ID Token validation steps.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified. You seem to be missing one or more of these checks. You may use an SDK-implemented validation if this implements all these checks. Otherwise it is recommended to do these comparisons yourself.

```
                // todo
```

MISSING_VERIFY_ID_TOKEN: Missing validation of ID Token.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).

You seem to be missing such validation in the code locations where you implement the token request flow.

```
            There are five values in the ID token response that must be verified.
            You may use an SDK-implemented validation if this implements all these checks.
            Otherwise it is recommended to do these comparisons yourself.
```

MISSING_VERIFY_NONCE: Missing verify Nonce

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). The nonce is one of the required values made to protect against replay attacks in token responses. The nonce value serves pretty much the same purpose for the ID token response as State does for the authorization response. Add nonce to your authentication request and store the value. Check the nonce claim of the ID token against the stored value for validation. Error 401 must be returned if nonces do not match.

```
            // todo
```

MISSING_VERIFY_TOKEN_ISS: Missing verify iss parameter in ID Token

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). iss parameter validation

```
            // todo
```

MISSING_VERIFY_TOKEN_SIGN: Missing verify cryprographic signature in ID token

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). Cryptographic validation

```
            // todo
```

USING_INCOMPLETE_ID_TOKEN_VALIDATOR: Using incomplete SDK-implemented ID Token validation.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). You seem to be using an ID token validation method implemented by an SDK which is known to be incomplete, as it does not implement all five checks. There are five values in the ID token response that must be verified. In Google's APIs for example, the com/google/api/client/auth/openidconnect/IdTokenVerifier fails to implement the checks for validating the Key signatures and Nonce. Meanwhile the com.google.api.client.googleapis.auth.oauth2.GoogleIdTokenVerifier implements crypto signature validation. Still it misses Nonce check.(https://github.com/googleapis/google-api-java-client/blob/master/google-api-client/src/main/java/com/google/api/client/googleapis/auth/oauth2/GoogleIdTokenVerifier.java) Otherwise it is recommended to do these comparisons yourself.

```
                // todo
```

B.2 Raw data evaluation: Atricore [7]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/Oidc-FindSecbugs-Eval/Eval/atricore/target/atricore-1.0-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	1	0.00
Medium Priority Warnings		0.00
Total Warnings	**1**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

- Security Warnings
- Details

Summary

Warning Type	Number
Security Warnings	1
Total	**1**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code	Warning
SECMVIDT	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo2. Bug type MISSING_VERIFY_ID_TOKEN (click for details) In class idbus.oidc.GoogleAuthzTokenConsumerProducer In method idbus.oidc.GoogleAuthzTokenConsumerProducer.doProcessAuthzTokenResponse(AuthorizationCodeResponseUrl) At GoogleAuthzTokenConsumerProducer.java:[lines 58-226]

Details

MISSING_VERIFY_ID_TOKEN: Missing validation of ID Token.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).

You seem to be missing such validation in the code locations where you implement the token request flow.

```
There are five values in the ID token response that must be verified.
You may use an SDK-implemented validation if this implements all these checks.
Otherwise it is recommended to do these comparisons yourself.
```

B.3 Raw data evaluation: Firebase [35]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/Oidc-FindSecbugs-Eval/Eval/firebase/target/firebase-1.0-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	3	0.00
Medium Priority Warnings	7	0.00
Total Warnings	**10**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

- Security Warnings
- Details

Summary

Warning Type	Number
Security Warnings	10
Total	**10**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code	Warning
SECIIDTV	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo1. Bug type INCOMPLETE_ID_TOKEN_VERIFICATION (click for details) In class firebase.filetobeanalyzed.FirebaseTokenUtils In method firebase.filetobeanalyzed.FirebaseTokenUtils.newIdTokenVerifier(Clock, String, String) At FirebaseTokenUtils.java:[line 117]
SECIIDTV	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo1. Bug type INCOMPLETE_ID_TOKEN_VERIFICATION (click for details)

In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.getErrorIfContentInvalid(IdToken)
At FirebaseTokenVerifierImpl.java:[lines 171-216]

SECITVCF When performing an ID token check the specification requires that you return HTTP 401.

Bug type IMPROPER_TOKEN_VERIFY_CONTROL_FLOW (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.getErrorIfContentInvalid(IdToken)
Called method com.google.api.client.auth.openidconnect.IdToken.verifyAudience(Collection)
At FirebaseTokenVerifierImpl.java:[lines 171-216]

SECITVCF When performing an ID token check the specification requires that you return HTTP 401.

Bug type IMPROPER_TOKEN_VERIFY_CONTROL_FLOW (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.getErrorIfContentInvalid(IdToken)
Called method com.google.api.client.auth.openidconnect.IdToken.verifyIssuer(Collection)
At FirebaseTokenVerifierImpl.java:[lines 171-216]

SECITVCF When performing an ID token check the specification requires that you return HTTP 401.

Bug type IMPROPER_TOKEN_VERIFY_CONTROL_FLOW (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.isSignatureValid(IdToken)
Called method com.google.api.client.auth.openidconnect.IdToken.verifySignature(PublicKey)
At FirebaseTokenVerifierImpl.java:[lines 231-236]

SECMVNONCE ID Tokens must be validated by the Relying Party (Client). The nonce is a cryptographically opaque value like the state value which binds an authentiaction request to the ID Token

Bug type MISSING_VERIFY_NONCE (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenUtils
In method firebase.filetobeanalyzed.FirebaseTokenUtils.newIdTokenVerifier(Clock, String, String)
At FirebaseTokenUtils.java:[line 117]

SECMVNONCE ID Tokens must be validated by the Relying Party (Client). The nonce is a cryptographically opaque value like the state value which binds an authentiaction request to the ID Token

Bug type MISSING_VERIFY_NONCE (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.getErrorIfContentInvalid(IdToken)
At FirebaseTokenVerifierImpl.java:[lines 171-216]

SECMVTEXP ID Tokens must be validated by the Relying Party (Client). Freshness validations

Bug type MISSING_VERIFY_TOKEN_EXP (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenUtils
In method firebase.filetobeanalyzed.FirebaseTokenUtils.newIdTokenVerifier(Clock, String, String)
At FirebaseTokenUtils.java:[line 117]

SECMVTSIGN ID Tokens must be validated by the Relying Party (Client). Cryptographic validation

Bug type MISSING_VERIFY_TOKEN_SIGN (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenUtils
In method firebase.filetobeanalyzed.FirebaseTokenUtils.newIdTokenVerifier(Clock, String, String)
At FirebaseTokenUtils.java:[line 117]

SECREQTVER Token validation requires proper control flow. You seems to have reversed the boolean of one of your checks.

Bug type REVERSED_IF_EQUALS_ID_TOKEN_VERIFY (click for details)
In class firebase.filetobeanalyzed.FirebaseTokenVerifierImpl
In method firebase.filetobeanalyzed.FirebaseTokenVerifierImpl.isSignatureValid(IdToken)
Called method com.google.api.client.auth.openidconnect.IdToken.verifySignature(PublicKey)
At FirebaseTokenVerifierImpl.java:[lines 231-236]

Details

INCOMPLETE_ID_TOKEN_VERIFICATION: Missing one or more of ID Token validation steps.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified. You seem to be missing one or more of these checks. You may use an SDK-implemented validation if this implements all these checks. Otherwise it is recommended to do these comparisons yourself.

```
                // todo
```

IMPROPER_TOKEN_VERIFY_CONTROL_FLOW: Token validation control flow.

Any check failing must lead to a HTTP 401 response for proper control flow for token verification.

MISSING_VERIFY_NONCE: Missing verify Nonce

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). The nonce is one of the required values made to protect against replay attacks in token responses. The nonce value serves pretty much the same purpose for the ID token response as State does for the authorization response. Add nonce to your authentication request and store the value. Check the nonce claim of the ID token against the stored value for validation. Error 401 must be returned if nonces do not match.

```
            // todo
```

MISSING_VERIFY_TOKEN_EXP: Missing verify freshness of ID token

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). Freshness validation

```
            // todo
```

MISSING_VERIFY_TOKEN_SIGN: Missing verify cryprographic signature in ID token

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). Cryptographic validation

```
            // todo
```

REVERSED_IF_EQUALS_ID_TOKEN_VERIFY: Token validation reverse equals in if check.

Remember that any failed verificaton of ID token is that idToken.verifyx() is false. Therefore a good patterns is a chain of if/else conditionals with if(!verifyx()) --> unauhorize

```
              if(!oidcConfig.nonce.equals(idToken.getPayload().getNonce())) {
                return Response.status(Response.Status.UNAUTHORIZED)
                        .entity("The provided nonce did not match the one saved from the authorization request.")
                        .build();
            }
            if(!idToken.verifySignature(publicKey)){
                return Response.status(Response.Status.UNAUTHORIZED)
                        .entity("The jwt signature is not valid.")
                        .build();
            }
            if(!idToken.verifyAudience(Collections.singleton(config.getProperty("clientId")))) {
                return Response.status(Response.Status.UNAUTHORIZED)
                        .entity("This request does not seem like it was meant for this audience.")
                        .build();
            }
            if(!idToken.verifyExpirationTime(Instant.now().toEpochMilli(), DEFAULT_TIME_SKEW_SECONDS)){
                return Response.status(Response.Status.UNAUTHORIZED)
                        .entity("Token expired.")
                        .build();
            }
            if(!idToken.verifyIssuer(String.valueOf(providerMetadata.get("issuer")))) {
                return Response.status(Response.Status.UNAUTHORIZED)
                        .entity("The expected issuer did not match.")
                        .build();
            }
```

```
// .... other checks
authorizationCodeFlow.createAndStoreCredential(tokenResponse, oidcConfig.appuuid.toString());

return Response.ok()
        .entity(tokenResponse)
        .build();
```

B.4 Raw data evaluation: SonarQube [94]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/evaluation-files/sonar-auth-oidc/target/sonar-auth-oidc-plugin-2.0.1-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 3 packages.

Metric	Total	Density*
High Priority Warnings	10	0.00
Medium Priority Warnings	1	0.00
Total Warnings	**11**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

- Security Warnings
- Details

Summary

Warning Type	Number
Security Warnings	11
Total	**11**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code	Warning
SECISAUTH	The new update of the OAuth 2.0 standard disallows usage of this method entirely: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-13#section-3.4 Bug type USING_PASSWORD_GRANT_OAUTH (click for details)

In class com.nimbusds.oauth2.sdk.ResourceOwnerPasswordCredentialsGrant
In method com.nimbusds.oauth2.sdk.ResourceOwnerPasswordCredentialsGrant.parse(Map)
At ResourceOwnerPasswordCredentialsGrant.java:[line 192]

SECMVIDT According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo2.

Bug type MISSING_VERIFY_ID_TOKEN (click for details)
In class org.vaulttec.sonarqube.auth.oidc.OidcClient
In method org.vaulttec.sonarqube.auth.oidc.OidcClient.getTokenResponse(AuthorizationCode, String)
At OidcClient.java:[lines 151-161]

SECMVIDT According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo2.

Bug type MISSING_VERIFY_ID_TOKEN (click for details)
In class org.vaulttec.sonarqube.auth.oidc.OidcClient
In method org.vaulttec.sonarqube.auth.oidc.OidcClient.getUserInfo(AuthorizationCode, String)
At OidcClient.java:[lines 112-146]

SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.

Bug type MISSING_VERIFY_OIDC_STATE (click for details)
In class com.nimbusds.openid.connect.sdk.AuthenticationResponseParser
In method com.nimbusds.openid.connect.sdk.AuthenticationResponseParser.parse(HTTPRequest)
At AuthenticationResponseParser.java:[line 295]

SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.
SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.
SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.
SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.
SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.
SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.

Bug type MISSING_VERIFY_OIDC_STATE (click for details)
In class com.nimbusds.openid.connect.sdk.AuthenticationResponseParser
In method com.nimbusds.openid.connect.sdk.AuthenticationResponseParser.parse(URI, Map)
At AuthenticationResponseParser.java:[line 71]

SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.

Bug type MISSING_VERIFY_OIDC_STATE (click for details)
In class org.vaulttec.sonarqube.auth.oidc.OidcClient
In method org.vaulttec.sonarqube.auth.oidc.OidcClient.getAuthorizationCode(HttpServletRequest)
At OidcClient.java:[lines 93-108]

Details

USING_PASSWORD_GRANT_OAUTH: Usage of insecure authorization grant. Use redirection flow instead.

Instead of the password grant, use proper redirect methods with for example authorization code grant.

```
AuthorizationCode code = new AuthorizationCode("xyz...");
 URI callback = new URI("https://client.com/callback");
 AuthorizationGrant codeGrant = new AuthorizationCodeGrant(code, callback);

 // The credentials to authenticate the client at the token endpoint
 ClientID clientID = new ClientID("123");
 Secret clientSecret = new Secret("secret");
 ClientAuthentication clientAuth = new ClientSecretBasic(clientID, clientSecret);

 // The token endpoint
```

```
URI tokenEndpoint = new URI("https://c2id.com/token");

// Make the token request
TokenRequest request = new TokenRequest(tokenEndpoint, clientAuth, codeGrant);

TokenResponse response = OIDCTokenResponseParser.parse(request.toHTTPRequest().send());
```

MISSING_VERIFY_ID_TOKEN: Missing validation of ID Token.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).

You seem to be missing such validation in the code locations where you implement the token request flow.

```
There are five values in the ID token response that must be verified.
You may use an SDK-implemented validation if this implements all these checks.
Otherwise it is recommended to do these comparisons yourself.
```

MISSING_VERIFY_OIDC_STATE: State verification check is missing in your handling of authorization code flow response from IdP.

Remember to check that the state matches to avoid CSRF attacks.

```
if(!successResponse.getState().equals(state)) {
  // Unauthorized
}
```

B.5 Raw data evaluation: Liferay [52]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/Oidc-FindSecbugs-Eval/Eval/liferay/target/liferay-1.0-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	1	0.00
Medium Priority Warnings		0.00
Total Warnings	**1**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

Summary

Warning Type	Number
Security Warnings	1
Total	**1**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code **Warning**

SECVMOS The state parameter in the Authentication Response must be verified for checking integrity of the IdP.

Bug type MISSING_VERIFY_OIDC_STATE (click for details)
In class portal.oidc.OpenIdConnectServiceHandlerImpl
In method
portal.oidc.OpenIdConnectServiceHandlerImpl.getAuthenticationSuccessResponse(HttpServletRequest)
At OpenIdConnectServiceHandlerImpl.java:[lines 278-310]

Details

MISSING_VERIFY_OIDC_STATE: State verification check is missing in your handling of authorization code flow response from IdP.

Remember to check that the state matches to avoid CSRF attacks.

```
if(!successResponse.getState().equals(state)) {
  // Unauthorized
}
```

B.6 Raw data evaluation: Codice/ddf [14]

FindBugs Report

Project Information

Project:

FindBugs version: 3.0.1

Code analyzed:

- /home/elias/git/masterthesis/new-findsecbugs/Oidc-FindSecbugs-Eval/Eval/ddf/target/ddf-1.0-SNAPSHOT.jar

Metrics

0 lines of code analyzed, in 0 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings	1	0.00
Medium Priority Warnings		0.00
Total Warnings	**1**	**0.00**

(Defects per Thousand lines of non-commenting source statements)*

Contents

- Security Warnings
- Details

Summary

Warning Type	Number
Security Warnings	1
Total	**1**

Warnings

Click on a warning row to see full context information.

Security Warnings

Code	Warning
SECMVIDT	According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client). There are five values in the ID token response that must be verified todo2. Bug type MISSING_VERIFY_ID_TOKEN (click for details) In class oidc.resolver.OidcCredentialsResolver In method oidc.resolver.OidcCredentialsResolver.getOidcTokens(AuthorizationGrant, OIDCProviderMetadata, ClientAuthentication, int, int) At OidcCredentialsResolver.java:[lines 212-231]

Details

MISSING_VERIFY_ID_TOKEN: Missing validation of ID Token.

According the OpenID Connect specification, ID Tokens must be validated by the Relying Party (Client).

You seem to be missing such validation in the code locations where you implement the token request flow.

```
There are five values in the ID token response that must be verified.
You may use an SDK-implemented validation if this implements all these checks.
Otherwise it is recommended to do these comparisons yourself.
```

Elias Brattli Sørensen

Using Static Analysis to Detect Vulnerabilities in OpenID Connect Clients

NTNU
Kunnskap for en bedre verden