Proof-of-Possession Tokens in Microservice Architectures

Ruaridh Watt

January 26, 2018

MSc COMPUTER SCIENCE THESIS

External Partner: Omegapoint
External Supervisors: Markus Örebrand & Björn Nyberg
Internal Supervisor: Mikael Rännar
Examiner: Henrik Björklund

Abstract

The popular OAuth 2.0 Framework specifies the use of Bearer Tokens for the transmission of authorization credentials. A Bearer Token has the property that any party in possession of it can use the it. Requests including Bearer Tokens are therefore performed over a secure channel to ensure that tokens cannot be obtained by eavesdroppers.

There are, however, still ways in which tokens may be leaked, including cross-site scripting and man-in-the-middle attacks. In situations where Bearer Tokens do not provide adequate security, proof-of-possession techniques may be employed to bind tokens to clients, thereby mitigating token leakage.

This project presents a method which can be used to bind tokens to clients based on authentication performed by an external identity provider. How clients form proof-of-possession tokens is also described. The result is a token which may be used to transmit authorization credentials over an insecure channel.

Token performance is measured in terms of client key generation time, token generation time, and authorization time. The effect different signing algorithms have on performance is measured and the proof-of-possession token is compared to Bearer Token based authorization.

The results show that Elliptical Curve cryptography may be employed by lightweight devices to reduce client key and token generation times. The analysis also shows that an increase in authorization time by a factor of between six and nine can be expected when compared to Bearer Token based authorization.
**Abbreviations**

**API**  
Application Programming Interface  
A communication specification which defines how a software component may be interacted with.

**CQRS**  
Command-query responsibility segregation  
An architectural pattern used in microservice architectures in which query and update capabilities are split into two separate services.

**CSRF**  
Cross-site request forgery  
An attack in which users are forced or tricked into issuing requests to a web-application they are already signed into.

**HTTP**  
Hypertext Transfer Protocol  
An application layer, text based, request-response style protocol used in the transfer of hypertext.

**IANA**  
Internet Assigned Numbers Authority  
The authority responsible for the global coordination of the DNS Root, IP addressing, and other Internet protocol resources. Protocol registries are provided in coordination with the IETF.

**IETF**  
Internet Engineering Task Force  
An organization consisting of over 100 Working Groups which aim to produce high quality, relevant technical documents to “make the Internet work better”. Publications are referred to as RFCs (request for comments) and fall into three main categories: Proposed Standards, Internet Standards and Best Common Practices.

**JSON**  
JavaScript Object Notation  
A lightweight data-interchange format based on a subset of the JavaScript programming language.

**JWK**  
JSON Web Key  
A standardized means of representing a cryptographic key using JSON.

**JWT**  
JSON Web Token  
A compact, URL-safe token format used to transmit claims.

**MAC**  
Message Authentication Code
A data string used to authenticate a message. Computed and verified using a single shared secret key. MACs guard against message modification and serve as proof-of-possession of the secret key.

**MITM**  Man-in-the-middle
An attack in which communication between two systems is intercepted. The attacker acts as a proxy and is able to read, insert and modify communications.

**PoP**  Proof-of-possession
Verifiable proof that an entity is privy to a secret that does not disclose the secret itself.

**SAML**  Security Assertion Markup Language
An XML based token format used to securely transmit identity and authorization information.

**TLS**  Transport Layer Security
A transport layer protocol used to provide privacy and data integrity between two communicating applications. Made up of the TLS Handshake Protocol and the TLS Record Protocol.

**XSS**  Cross-site scripting
An attack in which malicious script is injected into a website.
1 Introduction

The microservice architecture has emerged as a set of common software development practices employed by a number of leading organizations. By focusing on system reliability, scalability, replaceability and reducing the time-to-market of new releases, a number of large systems have evolved into microservices [28].

Systems based on a microservice architecture commonly expose functionality to front-end applications through an API [35]. Securing an API involves binding authorization credentials to requests. How and if a request is processed depends on the authorization credentials included in the request.

Front-end applications typically obtain a user’s authorization credentials as a result of the user signing in. By signing in, the user effectively delegates access to the front-end application. Before authorization credentials are transferred to the front-end application, the user must first be authenticated.

A system may choose to perform user authentication itself, however, due to the risks involved in storing user credentials and the work involved in implementing sign-up, sign-in, and password recovery, many applications are choosing to outsource user management to external identity providers [27]. Many popular online services can also act as identity providers, including Google, Facebook, Twitter, Github and many more [2].

When a user signs in, the front-end application may be issued with a key or token which enables resource servers to obtain the user’s authorization credentials. Keys enable resource servers to look up authorization credentials whereas tokens contain the authorization credentials themselves. Tokens prove to be particularly suitable in microservice architectures as they enable services to act autonomously: an authorization decision can be made without the need to fetch authorization credentials.

The OAuth 2.0 Framework [15] is billed as “the industry-standard protocol for authorization” [31]. In OAuth 2.0, authorization credentials are transmitted as Access Tokens which are defined in the framework as “a string representing an access authorization issued to the client” [15]. At the time of writing, there is only one registered type of Access Token that may be returned in the OAuth 2.0 Framework: the Bearer Token [32].

Bearer Tokens are defined in the framework as

“A security token with the property that any party in possession of the token (a “bearer”) can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession)” [22].

Bearer Tokens are very convenient from a usage standpoint: requests may be authorized by simply including the token in the request. The drawback is that Bearer Tokens must be kept secret as they can be used by anyone in possession of them to gain access to protected resources.
Another option is to require the bearer to “prove possession of cryptographic key material (proof-of-possession)”. Proof-of-possession can be used to render leaked tokens useless, protect the integrity of requests, and enable replay protection.

This project examines how authentication performed by an external identity provider can be used to secure an API exposed by a microservice architecture. The solution utilizes Access Tokens which require the bearer to provide proof-of-possession. The feasibility of the solution is examined through implementation and performance testing.

1.1 Problem Statement

There are two problems addressed in this project. The first involves the use of an identity provider. There are a number of different ways in which an OpenID Connect compliant identity provider may be used [20]. Proper identity provider usage is essential to the overall security of a system.

The second problem involves the vulnerabilities associated with Bearer Tokens. Bearer token security relies on a secure channel to provide protection from token disclosure, however there are still a number of ways in which bearer tokens may be leaked, including cross-site scripting (XSS) and man-in-the-middle attacks (MITM) [5, 7].

1.2 Project Goals

The project goals attempt to address the problems stated above by specifying appropriate identity provider usage and an Access Token type that guards against token leakage. In order to analyze the feasibility of the proposals in a microservice environment an implementation and testing phase is also required. The goals can be summarized as follows:

1. Define an Access Token type which guards against token leakage and protects against request modification and replay through proof-of-possession.

2. Specify a protocol which can be used to issue Access Tokens to front-end applications based on authentication performed by an OpenID Connect compliant identity provider.

3. Evaluate the feasibility of the proposals through implementation and performance testing.

1.3 Structure

Firstly, a brief background in microservice architectures, OAuth 2.0, OpenID Connect and Access Token formats is given, followed by a summary of previous and ongoing work in this area. The solution is then presented in the form of an Access Token specification, system design, and protocol. Finally the implementation and test results are presented together with the implications they have on the feasibility of the proposed solution.
2. Background

As this project is concerned with securing microservices, using a method built upon the OAuth 2.0 framework, an introduction to these areas is provided. OpenID Connect, a widely used OAuth 2.0 extension, is also introduced as it is used in interactions with the identity provider. Finally, an introduction to the token format which will be used in this project is given.

2.1 Microservice Architecture

A microservice architecture is an architectural style in which an application is composed of a collection of loosely coupled, autonomous services that can be independently developed, deployed and scaled. In order to act autonomously services must own their own data. This means that each service has exclusive access to its own data store. Inter-service communication is conducted asynchronously via a message broker. This maintains loose coupling and increases fault tolerance. Routing and service orchestration is performed by API Gateways in order to hide the granularity of the underlying microservices and provide suitable APIs for a variety of clients.

Other design choices may include modeling data using Event Sourcing and separating data-update and data-querying capabilities with Command-query responsibility segregation (CQRS) [28, chapter 5]. This involves modeling data as a sequence of state changes or events. A dedicated reporting service can subscribe to events published by multiple other services and build an aggregate model of the current state. This can be used to hide the granularity of the underlying services and maintain loosely coupled services.

Securing autonomous, loosely coupled microservices is typically done with an API Gateway [28, Chapter 6]. This project, however, explores the possibility of request authorization performed independently by individual service instances.

2.2 OAuth 2.0

Prior to 2006 it was common for third-party applications to request the user’s authentication credentials for a service to enable delegated access [35]. For example, a printing service would require a user to submit their username and password for a photo hosting service in order to allow them access to their photos. This method of access delegation does not enable users to limit access; the printing service is able to perform all the operations of the account owner.

OAuth 2.0, billed as “the industry-standard protocol for authorization” [31], effectively decouples authentication and authorization. This enables users to grant third-party applications limited access to services on their behalf without disclosing their authentication credentials to the third-party.

The OAuth 2.0 framework describes a system comprising of a Resource Owner, Client, Authorization Server, and Resource Server as described in Table 2.
2. BACKGROUND

| Resource Owner                | An end-user or other entity that has some rights or privileges to a particular resource. Capable of authenticating itself and granting access to Clients. |
| Client                        | An application that wishes to perform resource requests on behalf of the Resource Owner. |
| Resource Server               | A server holding protected resources. Capable of processing and responding to resource requests with respect to an Access Token included in the request. |
| Authorization server          | A server which issues Access Tokens to clients as a result of successful authentication and access grant from the Resource Owner. |

Table 2: OAuth 2.0 Entities

When a Client wishes to access protected resources on behalf of an end-user it redirects the end-user to the Authorization Server with an Authorization Request via a web-browser. The end-user authenticates, authorizes the Client’s request and is then redirected back to the Client with an Authorization Grant, as shown in Figure 1.

Figure 1: OAuth 2.0 Authorization Grant

The Authorization Grant may include an Access Token (Implicit Grant) or an Authorization Code (Code Grant) which can be exchanged for an Access Token. The Code Grant is a more secure method of issuing a Client with an Access Token as it requires Client authentication which guards against Authorization Grant interception. This method is, however, only suitable for Clients capable of maintaining the confidentiality of their credentials, for example a server-side application.

In addition to Access Tokens, Authorization Servers may issue Refresh Tokens.
2. BACKGROUND

Refresh Tokens can be used by Clients to obtain a new Access Token from the Authorization Server. If an Access Token includes an expiry time, Clients may use a Refresh Token to prolong their access to protected resources without requiring re-authentication of the Resource Owner.

2.3 OpenID Connect

OpenID Connect extends the OAuth 2.0 framework to include an identity layer. The capabilities of the Authorization Server are extended to enable the issuing of Identity Tokens. Identity Tokens enable Clients to obtain and verify identity and profile information of the end-user. These may be used to verify authentication performed by an identity provider.

OpenID Connect also adds a Hybrid Flow and extended grant types. This enables Authorization Servers to issue an Authorization Code for server-side applications and tokens for front-end application simultaneously.

2.4 Token Format

Tokens are used to securely transmit information between parties. The information is protected from modification by a digital signature, message authentication code, or encryption. The integrity and origin of the information can be verified using a key obtained from the issuer.

There are many token formats in use today. Many organizations use a proprietary format developed to meet their specific requirements. There are, however, also a number of public standards which may be used.

Two widely used and well documented standard token formats are the Security Assertion Markup Language (SAML) [30] and JSON Web Tokens [3]. SAML is an XML-based markup language while JSON Web Tokens use JavaScript Object Notation (JSON).

XML is a powerful language which incorporates validation, query and transformation capabilities, however, when it comes to the transmission of immutable information between parties, the lightweight JSON data format, together with the compact, url-safe JSON Web Token standard proves to be particularly suited to the OAuth 2.0 framework. Formatting tokens using JSON also fits in with the growing trend of APIs using only JSON as the data format for requests and responses [35].

2.4.1 JSON Web Token

A JSON Web Token (JWT) is a compact, url safe means of transmitting claims between two parties. The claims are contained in a JSON object which constitutes the token payload.

The payload typically contains claims pertaining to the subject, issuer, audience and token expiration. Such claims are registered with the Internet Assigned
Numbers Authority (IANA) and have well defined semantics, however implementation specific claims may also be included. The full list of registered claims can be viewed in the IANA registry [19].

JWTs can be protected by a digital signature (asymmetric cryptography) or message authentication code (symmetric cryptography). This enables the integrity and origin of the claims to be verified. Message authentication codes are computed and verified using a single shared secret key. Digital signatures are computed using a private key and verified using the corresponding public key. Using digital signatures enables the separation of JWT issuing and verification capabilities, avoids the problem of securely distributing shared secrets, and enables JWT verification to be performed by untrusted clients. This project will therefore only consider signed JWTs.

A signed JWT consists of a header, payload and signature in the following format:

```
<header>.<payload>.<signature>
```

Base64URL encoding [14, Section 5] is used to serialize each part as a compact, url-safe string.

The header contains claims that describe the cryptographic operations applied to the payload. The only mandatory claim is the algorithm used to compute the signature. Table 3 shows the signing algorithms that will be used in this project.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS256</td>
<td>RSASSA-PKCS1-v1_5 using SHA-256</td>
</tr>
<tr>
<td>RS384</td>
<td>RSASSA-PKCS1-v1_5 using SHA-384</td>
</tr>
<tr>
<td>RS512</td>
<td>RSASSA-PKCS1-v1_5 using SHA-512</td>
</tr>
<tr>
<td>ES256</td>
<td>ECDSA using secp256r1 and SHA-256</td>
</tr>
<tr>
<td>ES384</td>
<td>ECDSA using secp384r1 and SHA-384</td>
</tr>
</tbody>
</table>

Table 3: The signing algorithms used in this project

Detailed information on the computation and verification of the signing algorithms can be found in the JSON Web Algorithms specification [17].

Given a header, payload and secret key, a signed JWT can be composed using
Algorithm 1

**Data:** A header, payload, and private key, \( K \)

**Result:** A digitally signed JWT

**Function** `composeJWT(header, payload, K)

encoded ← base64url(header) + "." + base64url(payload);
signature ← sign(encoded, K);
jwt ← encoded + "." + base64url(signature);
return jwt end

Algorithm 1: JWT composition

There are a large number of libraries available which facilitate the composition and verification of JWTs, many of which are published on the [jwt.io](http://jwt.io) website.

To enable Resource Servers to verify JWTs issued by an Authorization Server the public keys used to sign JWTs must be published. This project uses the JSON Web Key (JWK) format in public key distribution [16].

A JWT consisting of authorization credentials signed by the token issuer is an example of a Bearer Token. JWTs may, however, include a confirmation claim as specified in Proof-of-Possession Key Semantics for JSON Web Tokens [18]. The confirmation claim is used to bind a JWT to a Client and verify Client proof-of-possession.

### 2.4.2 JWT Confirmation Claim

Use of the JWT Confirmation claim is defined in Proof-of-Possession Key Semantics for JSON Web Tokens [18]. This specification defines how Client keys may be included in JWTs. This effectively binds a JWT to a specific Client by making it unusable without the Client’s secret key.

When a Client requests access to protected resources on behalf of an end-user it can include its public key in the request. After the end-user authenticates and authorizes the Client’s request, a JWT including the Client’s public key can be issued.

The JWK confirmation method [18, Section 3.2] is used to include a Client’s public key in a JWT in JWK format. An example of this is shown in Listing 1. In this example the Client has submitted an Elliptical Curve key [16] and will use the EC256 algorithm to prove possession of the corresponding private key, as specified in the JWK’s `alg` claim.

If a confirmation method is included in a JWT, the Client must provide proof-of-possession with each request. There is no accepted standard which specifies how Clients should provide proof-of-possession, however inspiration may be taken from related work in this field.
2. BACKGROUND

Listing 1: An example JWT payload including a JWK confirmation method

```json
{
    "sub": "123456789",
    "cnf": {
        "jwk": {
            "alg": "EC256",
            "crv": "P-256",
            "kty": "EC",
            "use": "sig",
            "x": "18wHLeIgW9wVN6VD1Tgxy2LszYkMf6J8njVAibvhM",
            "y": "-V4dS4uLMgp_4fY4j8ir7cl1TXlFdAgcM5S07KcSA"
        }
    }
}
```
3 Related Work

The concept of proof-of-possession is not itself a new idea, in fact, a response digest is used in HTTP Digest Authentication \[12\] to prove possession of the user’s password without the need to transmit the password itself over the wire. In this challenge-response protocol the MD5 hash function is typically used to distill security and message related data into a response which can be verified by the challenging server, providing it also has some knowledge of the user’s password.

3.1 SAML Holder of Key Assertion Profile

SAML includes a Holder-of-Key Assertion Profile \[29\]. This specifies how a token issuer can embed X.509 data from an X.509 certificate held by the attesting entity. The attesting entity submits the assertion and X.509 certificate along with proof-of-possession of the corresponding private key. This allows the relying party to confirm the attesting entity. The details of how the attesting entity proves possession of the private key is, however, not specified.

This is very similar to the JWK confirmation method for JWTs and may well have been the inspiration behind this confirmation method.

3.2 Kerberos Tickets

Kerberos \[13\] uses a system of shared symmetric keys to enable authentication over an insecure network. Kerberos tickets are used to securely transmit claims over the network. A conceptual overview of how tickets are bound to clients follows.

Clients begin by requesting credentials for a specific service from the authentication server (AS). The reply includes a ticket for use with the corresponding service (\(T_{srv}\)) encrypted with the service’s secret key (\(K_{srv}\)) and a session key (\(K_{ses}\)) encrypted with the clients secret key (\(K_{cli}\)). Only the service can decrypt \(T_{srv}\) and the session key can only be deciphered with a key derived from the user’s password.

Let \(E_K\) denote encryption with key \(K\). The AS response contains

\[
\{T_{srv}, E_{K_{cli}}(K_{ses})\}, \text{ where} \\
T_{srv} = E_{K_{srv}}(userID, K_{ses})
\]

The Client deciphers \(K_{ses}\) and uses it to encrypt an authenticator. The ticket and authenticator are then sent to the service:

\[
\{T_{srv}, E_{K_{ses}}(userID, timestamp)\}
\]

The service attempts to decipher the ticket, \(T_{srv}\). If successful, the ticket was created by the AS for this service and contains the ID and session key of the
user to which the ticket was issued. The service can then attempt to decipher the authenticator using the session key from the ticket. If successful, the Client has proven possession of the session key and it can be concluded that the ticket was issued to the Client.

Kerberos as a whole is, of course, slightly more complicated than this. The tickets contain values used to seed encryption and detect replay attacks. The system also specifies a Ticket Granting Service (TGS) which enables reduced transmission of data encrypted with a user’s secret key. Users first request a Ticket Granting Ticket (TGT) from the AS for use with the TGS. A session key is then used for further ticket requests aimed at the TGS.

Kerberos requires each service to have an associated symmetric key which are all stored on a central authentication server. The addition of services and the replication of the authentication server involves the distribution of shared secrets. This makes it unsuitable in dynamic environments where services may not be capable of establishing a secure connection to an authentication server.

3.3 Client Bound JWTs

There are two IETF drafts which outline methods to bind JWTs to Clients. The first draft, entitled OAuth 2.0 Message Authentication Code (MAC) Tokens [11], outlines an OAuth 2.0 token type extension: the MAC token. This token type requires the client to provide proof-of-possession of a shared symmetric key by computing a keyed message digest of a flexible set of parameters from the HTTP message. How the key is distributed to the client and resource servers is also outlined. This draft was last updated January 15, 2014 and has subsequently expired. Even though several questions remain in the draft, mac tokens have seen some limited use [35].

The second, much more recent draft is entitled OAuth 2.0 Proof-of-Possession (PoP) Security Architecture. Here an architecture is put forward in which a trusted third party binds symmetric or asymmetric client key information to the tokens issued. The draft refers, in turn, to Proof-of-Possession Key Semantics for JSON Web Tokens [18] as an example of how key information can be bound to a token and A Method for Signing HTTP Requests for OAuth [9] as an example of how clients can use a JWT structure to sign the token and, optionally, parts of the request. This draft is the closest relative to the proof-of-possession method presented in this project.

The latest work being done on token binding involves binding tokens to connections secured using Transport Layer Security (TLS) [10]. While this method is promising, additional challenges are posed by TLS Terminating Reverse Proxies [6] and non-TLS protected resources. For these reasons, and to keep transport layer and application layer security decoupled this project will only deal with purely application layer token binding techniques.
4 Method

The method used to achieve the project goals is given in this section. It comprises of a system design, token specification and protocol. These are used to issue Client bound Access Tokens based on authentication performed by an external OpenID Connect Provider. Clients are required to include their Access Token and proof-of-possession with each request. To achieve this, Clients compose a signed JWT containing the Access Token and data derived from the request. This is referred to as a PoP Token.

In the proposed system, authorization of requests is performed independently by individual service instances. Requests are only met if:

- The Access Token was issued by a trusted issuer.
- The Access Token has not expired.
- The Access Token contains sufficient privileges.
- The request originated from the Client to which the Access Token was issued.
- The request has not expired.
- The request has not been modified.
- The request has not been previously met.

In contrast to Bearer Tokens, disclosure of the tokens described here poses no threat. In addition, requests which include a PoP Token are protected from modification and replay even in the absence of TLS.

4.1 System Design

The system comprises of a set of microservices which communicate via a Message Broker using Event Sourcing as depicted in Figure 2.

An external OpenID Connect Provider is used to authenticate end-users and provide identity information to the Token Service. The Token Service issues Access Tokens to Clients based on this information. Clients submit authorization credentials and protect each request by including a PoP Token.

Figure 2 also shows the Key Events published by the Token Service. Each Token Service instance generates and rotates its own set of asymmetrical key-pairs. Private keys, which are used to sign Access Tokens, are never sent over the wire or written to disk. Whenever a Token Service is spawned or rotates its keys it emits a Key Event containing the public keys used to validate its signatures. Key Events are further described in Section 4.1.1.

The Key Set acts as the read-side of the Token Service, consuming Key Events and making the active public keys available as a JWK Set [16, Section 5]. This allows for the integration of external services which do not have access to the Message Broker. Services which are able to subscribe to Key Events maintain their own cache of active public keys.
4. METHOD

4.1.1 Key Event

A Key Event is published by a Token Service whenever its key set changes. A single Token Service instance may be capable of multiple different signing algorithms. A Key Event may therefore contain multiple public keys. Each key must be assigned an ID so that services can identify the key to use when verifying an Access Token signature.

Each Key Event also contains the key set’s expiry time as a Numeric Date [3, Section 2]. The expiry time is the time at which the Token Service will next rotate its keys plus the number of seconds its Access Tokens are valid for. For example, if a Token Service is spawned with a key rotation period of 30 days, and issues access tokens which are valid for 24 hours, the key set’s expiry time would be set to the current Numeric Date plus the number of seconds in 31 days. An example Key Event, in JSON, is shown in Listing 2.

4.2 Token Specifications

There are three token types used in the system: ID Tokens, Access Tokens and PoP Tokens, all of which are signed JWTs.

The claims used in ID Tokens are as specified in the OpenID Connect [20, Section 2]. Access and PoP Token claims are defined in the following two sections.

4.2.1 Access Token

Access Token claims are specified in Table 4 and an example Access Token payload is shown in Listing 3.
4. METHOD

```json
{
  "keys": [
    {
      "alg": "ES256",
      "crv": "P-256",
      "kid": "nhBNz5xpracw-9Ad2W7O-MEtYcv59Qpo0VVeCrH9GMA",
      "kty": "EC",
      "use": "sig",
      "x": "xZH7Gg9PF61I00_k4waYPoZh9x0bfqX-AmrjuWtjc",
      "y": "A6VQajFrcWjRQTepMH6jAtf-h9a9EWFcEvAnKmsH-us"
    },
    {
      "alg": "RS256",
      "e": "AQAB",
      "kid": "Avy407h0AL4chPT857s96kSK5rkcGMRXSq0N0pZmKf4",
      "kty": "RSA",
      "n": "xL01MFbUpgJ1mwdDyc_Tl2t2zEhKbLqLmrM3I6G0fUaiQS...",
      "use": "sig"
    }
  ],
  "exp": 1513091734
}
```

Listing 2: An example of a JSON serialized Key Event

<table>
<thead>
<tr>
<th>Claim</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aud</td>
<td>The audience for which the Access Token is valid.</td>
</tr>
<tr>
<td>sub</td>
<td>An identifier for the end-user, or subject.</td>
</tr>
<tr>
<td>iss</td>
<td>The token issuer.</td>
</tr>
<tr>
<td>cnf</td>
<td>The confirmation method. Must be an object containing a single public JWK: The Client’s public key.</td>
</tr>
<tr>
<td>exp</td>
<td>A Numeric Date specifying the token expiry time.</td>
</tr>
</tbody>
</table>

Table 4: Access Token claims
4. METHOD

Listing 3: An example Access Token payload

```json
{
  "aud": "api.rwatt.se",
  "sub": "auth0\{59b7c7d1cb7b3e63a1ae\}",
  "iss": "oauth.rwatt.se",
  "cnf": {
    "jwk": {
      "kty": "EC",
      "use": "sig",
      "crv": "P-256",
      "x": "nVRAdyxyZUKmH-1_GTtmUxs_HjDtaAJ0lQEKKHkTFm\T8",
      "y": "KSoOKOtAYsulF\S__d\mPnP\ByG8n88n-iGZECXpeA",
      "alg": "ES256"
    }
  },
  "exp": 1513177303
}
```

Additional claims may be included in the Access Token, for example, roles and/or groups may be used to grant additional privileges to specific users. Some OpenID Connect Providers support the customization of specific users’ ID Tokens. In this case additional privileges may be copied from the ID Token.

The Access Token header must include a *kid* claim specifying the ID of the key used to sign the token. This enables services to identify the key to use when verifying the token signature.

4.2.2 PoP Token

PoP Token claims are specified in Table 5 and an example PoP Token shown in Listing 4.

<table>
<thead>
<tr>
<th>Claim</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jti</td>
<td>The token ID.</td>
</tr>
<tr>
<td>at</td>
<td>The Client’s Access Token.</td>
</tr>
<tr>
<td>u</td>
<td>The URL to which the request will be sent, including query parameters.</td>
</tr>
<tr>
<td>m</td>
<td>The request method, e.g. POST.</td>
</tr>
<tr>
<td>b</td>
<td>The S256 digest of the request body, or the empty string for requests with no body content (See Figure 3).</td>
</tr>
<tr>
<td>exp</td>
<td>The Numeric Date specifying the token’s expiry time.</td>
</tr>
</tbody>
</table>

Table 5: The required PoP Token claims
4. METHOD

Listing 4: An example PoP Token payload

PoP tokens are signed JWTs containing the Client’s Access Token and request specific claims.

The Client’s signature provides proof-of-possession of the private key and protects the integrity of the token payload. By including request specific information in the token payload, services are able to verify that the request itself has not been modified.

The S256 digest of the request body is used to protect its integrity. This process is shown in Figure 3. The S256 definition used in this project is an extension of the S256 method used in Proof Key For Code Exchange [33].

A token identifier is also included to enable services to provide replay protection and the token is assigned an expiry time.

A PoP Token should not have to be valid for longer than 300 seconds. This provides enough time for the token to be processed (and allow for some clock skew) whilst minimizing the time a service must cache the token identifier in order to provide replay protection.
4. METHOD

```json
{
   "name": "Jane Doe",
   "email": "j.doe@rwatt.se"
}
```

Figure 3: S256 Digest of the Canonicalized body content
4.3 Sign-in

When an end-user signs in to a Client application an Access Token is issued to the Client allowing it to act on behalf of the end-user. The Token Service requires the identity of the end-user and the Client’s public key in order to compose an Access Token.

The Sign-in protocol defined in this section can be used to issue Access Tokens to Clients based on authorization performed by an OpenID Connect Provider. An overview of the Sign-in protocol in which the end-user authenticates via a web-browser is shown in Figure 4.

The Sign-in protocol begins with the Client issuing a request for the end-user to be authenticated by the OpenID Connect Provider. An Authorization Code is issued to the Client after successful authentication of the end-user. This part of the protocol conforms to the OpenID Connect Authorization Code Flow [20, Section 3.1] with a Proof Key For Code Exchange [33] and is described in Sections 4.3.1 to 4.3.3.

Typical OpenID Connect usage would see the Client exchange the Authorization Code for an ID Token. However, here, the code is submitted to the Token Service along with the Client’s public key. Once the Token Service exchanges the Authorization Code for an ID Token it holds the identity of the end-user and the Client’s public key. The Token Service can then return an Access Token to the Client. The protocol specification for this is defined in Sections 4.3.4 to 4.3.7.
4. METHOD

4.3.1 Authentication Request

An Authentication Request is issued by the Client to the OpenID Connect Provider via a web-browser. The HTTP GET method is used with request parameters serialized using URI Query String Serialization [20, Section 13.1].

In order to form an Authentication request the Client must generate and store two random strings of Unreserved Characters [4, Section 2.3]. The first is used as the state parameter, recording the Client’s state when the Authentication Request was sent. This will be returned by the OpenID Connect Provider as a parameter in the redirect URI of the Authorization Code Response. This enables the Client to detect redirects that do not originate from the OpenID Connect Provider, guarding against login Cross-Site Request Forgery (CSRF) [20].

The second random string is used as a code-verifier. The S256 digest of the code-verifier is used as a code-challenge. The OpenID Connect Provider links the code-challenge to the Authorization Code. This guards against the interception of the Authorization Code: The OpenID Connect Provider requires both the Authorization Code and the code-verifier in order to release the ID Token.

A full specification of the request parameters used is shown in Table 6 and an example Authentication Request is shown in Listing 5.

![Listing 5: An example Authentication Request](image)

4.3.2 Authentication and Authorization

The OpenID Connect Provider authenticates the end-user and asks the user for authorization. Authorization in this setting involves the end-user giving consent to the Token Service to access their identity information. The details of this stage depend on the OpenID Connect Provider used.

4.3.3 Authorization Code Response

After successful authentication and authorization the OpenID Connect Provider redirects the web-browser back to the Client with the Authorization Code and the state parameter in the URI fragment. At this point the Client verifies that
### 4. METHOD

<table>
<thead>
<tr>
<th>Request Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope</td>
<td>openid [profile] [email] [address] [phone]</td>
<td>The requested access privileges. Used to request that specific identity information be included in the ID Token.</td>
</tr>
<tr>
<td>response_type</td>
<td>code</td>
<td>Specifies that an Authorization Code be returned as a result of successful authentication.</td>
</tr>
<tr>
<td>response_mode</td>
<td>fragment</td>
<td>Specifies that the Authorization Code and state are returned as URI fragments instead of as query parameters. This avoids the parameters being sent on by the Client to the redirect URI.</td>
</tr>
<tr>
<td>client_id</td>
<td>&lt;client ID&gt;</td>
<td>The ID assigned to the Token Service by the OpenID Connect Provider. This is obtained by registering the Token Service as a client of the OpenID Connect Provider.</td>
</tr>
<tr>
<td>redirect_uri</td>
<td>&lt;redirect URI&gt;</td>
<td>The URI to which the web-browser will be redirected after successful authentication. Must be specified in the OpenID Connect Provider client account.</td>
</tr>
<tr>
<td>code_challenge</td>
<td>S256</td>
<td>The method used to compute the code_challenge from the code_verifier.</td>
</tr>
<tr>
<td>code_verifier</td>
<td>S256(code_verifier)</td>
<td>The S256 digest of the code_verifier. The code_verifier is a random string of 64 Unreserved Characters.</td>
</tr>
<tr>
<td>prompt</td>
<td>login</td>
<td>Optional. May be used to force re-authentication of the end-user. If specified, the possession of a Bearer Token, which may have been provided by the OpenID Connect Provider, will not be accepted as an authentication method.</td>
</tr>
</tbody>
</table>

Table 6: Authentication Request parameters
the state parameter returned matches the state supplied in the Authentication Request. This guards against CSRF attacks on the redirection endpoint [5, Chapter 9]. If the state parameter is missing, or differs from the stored state value, the Client discards the parameters contained in the URI fragment. An example Authorization Code Response is shown in Listing 6.

Example Authorization Code Response

```
HTTP/1.1 302 Found
Location: https://myapp.com/callback#code=nKtx8RD50EwEoNUr&,
         state=0865b13789843ca7
```

Listing 6: An example Authorization Code Response

### 4.3.4 Access Token Request

Once in possession of an Authorization Code the Client generates a new asymmetrical key pair. The private key must be stored in a secure manner to guard against unauthorized use. The Access Token Request contains the Client’s public key, the Authorization Code received from the OpenID Connect Provider, the redirect URI used, and the code-verifier. The code-verifier assures that the Access Token Request originates from the Client that issued the Authentication Request. This guards against Authorization Code interception. An example Access Token Request is shown in Listing 7.

Example Access Token Request

```
POST auth/token HTTP/1.1
Host: rwatt.se
Content-Type: application/json

{
    "code": "nKtx8RD50EwEoNUr",
    "codeVerifier": "yHbNpe4BJECdbEXTnZMpqJGWwpNNIwTZ8F84Xl9zofR807",
    "redirectUri": "https://myapp.com/callback",
    "jwk": {
        "alg": "ES256",
        "crv": "P-256",
        "kty": "EC",
        "use": "sig",
        "x": "jPgAr07df4ur2YM61PlR5sfnHUF30Iu8A-IsFv-Fd3E",
        "y": "P07I1lrKW2B3H3xaH8400z4adEFRzmAh6TUpvR8nluM"
    }
}
```

Listing 7: An example Access Token Request

### 4.3.5 ID Token Request

Upon receiving an Access Token Request, the Token Service uses the Authorization Code and code-verifier to form an ID Token Request. The parameters
used are shown in Table 7 and an example request is shown in Listing 8.

<table>
<thead>
<tr>
<th>Request Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code_verifier</td>
<td>&lt;code_verifier&gt;</td>
<td>The code-verifier from the Access Token Request.</td>
</tr>
<tr>
<td>redirect_uri</td>
<td>&lt;redirect URI&gt;</td>
<td>The redirect URI from the Access Token Request.</td>
</tr>
<tr>
<td>client_id</td>
<td>&lt;client ID&gt;</td>
<td>The client ID obtained by registering with the OpenID Connect Provider.</td>
</tr>
<tr>
<td>client_secret</td>
<td>&lt;client secret&gt;</td>
<td>The client secret obtained by registering with the OpenID Connect Provider.</td>
</tr>
<tr>
<td>grant_type</td>
<td>authorization_code</td>
<td>Specifies that the authorization grant was issued as an Authorization Code.</td>
</tr>
</tbody>
</table>

Table 7: ID Token Request parameters

<table>
<thead>
<tr>
<th>POST /token HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host: <a href="http://www.openidconnectprovider.com">www.openidconnectprovider.com</a></td>
</tr>
<tr>
<td>Content-Type: application/x-www-form-urlencoded</td>
</tr>
</tbody>
</table>

code=nKtx8RD5OeEoNUr&
code_verifier=yHbNpe4BJEcdbEXTnZMpqJGVpNNIwT89F84Xi9zoR8o7cC78B&
redirect_uri=http://myapp.com/callback&
client_id=XpBM6P1QpOv0qdM0ngfp4fedsro2s2858&
client_secret=TtLHEgDjGiWlxrVTz2XZ8-b1Tspr6X8oR07Vlpe5em3-qchxzvA&
grant_type=authorization_code

Listing 8: An example ID Token Request

4.3.6 ID Token Response

The exact format and content of the ID Token response depends on the OpenID Connect Provider. Other tokens may well be included in the response, however, since the OpenID Connect Provider is only being used as an identity provider, everything but the ID Token may be discarded.

A typical ID Token Response is shown in Listing 9.

An example of how the ID Token payload may look after decoding is shown in Listing 10.
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
    "access_token": "tor7f4elzGlzDRC9sD2t5EBmROeOnL01",
    "expires_in": 86400,
    "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
    "token_type": "Bearer"
}

Listing 9: An example ID Token Response

{
    "iss": "https://dv12rwt.eu.auth0.com/",
    "sub": "auth0|59b7c7d1ccbb73b1e63a1ea5",
    "aud": "XpBM6P10q0v0qdM0ngmp4fedsrbo2g50",
    "exp": 1513295493,
    "email": "u1@rwatt.se",
    "email_verified": true,
    "nickname": "u1",
    "name": "Joe Bloggs",
    "amr": [
        "mfa"
    ]
}

Listing 10: An example ID Token Payload
4.3.7 Access Token Response

Once the Token Service has received an ID Token it can generate an Access Token. The Access Token contains identity information from the ID Token and the Client’s public key as specified in Section 4.2.1.

The Access Token Response is a JSON object containing the Access Token, the token type identifier, and the length of time, in seconds, the Access Token is valid for. The token type identifier used is PoP. The ID Token may also be included to enable Clients to customize the user interface.

An example Access Token Response is shown in Listing 11.

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "eyJraWQiOiJ2dXhZLUpoMmI3VVN...",
  "token_type": "PoP",
  "expires_in": 86400,
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI...
}

Listing 11: An example Access Token Response

This concludes the Sign-in process. The Client is now able to generate PoP Tokens and include them in service requests.
4. METHOD

4.4 Service Request

Clients in possession of a valid Access Token can generate service requests. Service requests must include a PoP Token in the Authorization Header. This is accomplished by

1. Forming the request
2. Generating a PoP Token from the request and Access Token
3. Appending the PoP Token in the request’s Authorization Header.

Given the request in Listing 12 and an Access Token, a PoP Token can be generated.

```plaintext
POST /contact HTTP/1.1
Host: www.addressbook.com
Content-Type: application/json

{
  "name": "Jane Doe",
  "email": "j.doe@rwatt.se"
}
```

Listing 12: An example request

The S256 digest of the request body must be included in the PoP Token. If the body content is a JSON object it is recommended that the object be serialized using a canonical form prior to hashing. An example of this is shown in the PoP Token specification (Section 4.2.2). This enables repeatable hashing of the object, allowing for service implementations which deserialize the request prior to authorization.

The resulting PoP Token payload is shown in Listing 13.

```plaintext
{
  "jti": "3e6569ede9f6c585",
  "at": "eyJraWQiOiJBeHY0MDdoMEFMNGNoUFQ4NTdzOTZrU0s1cmtjR01...",
  "exp": 1513091734,
  "m": "POST",
  "u": "http://addressbook.com/contacts",
  "b": "X4V-o2FJH41ef8k7oMkREqljtPZc2XxtrW0X-XVHXE"
}
```

Listing 13: The PoP Token Payload for the request in Listing 12

The PoP Token is then signed by the Client and included in the Authorization header. The authorization type used is PoP. The final request is shown in Listing 14.
4. METHOD

![Listing 14: The request with PoP Token included in the Authorization header](image)

4.5 Authorization

Each request received by a service must pass authorization before being processed. Authorization involves the following sequence of procedures:

1. Parsing the Authorization Header
2. Access Token Verification
3. Access Token Validation
4. PoP Token Verification
5. PoP Token Validation
6. Access Control

There are libraries in a number of different programming languages that can be used to aid this process [3]. A general description of each procedure follows. Failure of any of the steps result in the request not being processed and an HTTP 403 Forbidden status being returned.

4.5.1 Parsing the Authorization Header

1. Check that the Authorization Method is PoP.
2. Deserialize the PoP Token.
3. Check that all of the required claims, as defined in Section 4.2.2, are present in the PoP Token payload.
4. Deserialize the Access Token.
5. Check that all of the required claims, as defined in Section 4.2.1, are present in the Access Token payload.

4.5.2 Access Token Verification

1. Get the key ID from the Access Token header (kid claim)
2. Check that the issuer claim specifies a trusted issuer (iss claim).
4. METHOD

3. Get the corresponding public JWK from the Key Set or the local cache of active public JWKs.

4. Check that the algorithm claim in the Access Token header is the same as the algorithm specified in the public JWK \[8\] (alg claim).

5. Verify the Access Token Signature using the public JWK.

4.5.3 Access Token Validation

1. Check that the Access Token has not expired (exp claim).

2. Check that the token is intended for use with this service (aud claim).

3. Check that the confirmation method is jwk (cnf claim).

4. Check that the JWK in the confirmation method specifies a single valid public JWK.

4.5.4 PoP Token Verification

1. Extract the JWK from the Access Token’s confirmation claim (cnf claim).

2. Check that the algorithm claim in the PoP Token header is the same as the algorithm specified in the JWK (alg claim).

3. Verify the PoP Token signature.

4.5.5 PoP Token Validation

1. Check that the PoP Token has not expired (exp claim).

2. Check that the request method is the same as the method claim (m claim).

3. Check that the request url is the same as the url claim (u claim).

4. Check that the S256 digest of the request body is the same as the body digest claim (b claim).

5. Extract the subject ID from the Access Token (sub claim).

6. Add the token identifier to the subject’s token cache until the token expires, failing if the cache already contains this identifier (jti claim).

4.5.6 Access Control

The service can then use the subject identifier and any additional claims included in the Access Token to check that the user issuing the request possesses adequate privileges. This may be as simple as checking that the subject is the owner of the resource, or may involve checking roles, groups or other Access Token claims. The exact method used in Access Control is left as an implementation detail.
5. Implementation

The method presented in Section 4 was implemented as a proof-of-concept and to judge the feasibility of PoP Token based authorization.

An example Client was implemented as an Angular 5 application for use with the back-end microservices implemented using the Lagom microservices framework. The implementation used Auth0 as the OpenID Connect Provider.

Table 8 provides some further information on the modules and libraries used in testing.

<table>
<thead>
<tr>
<th>Project Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>git.cs.umu.se/</td>
<td>An Angular 5 Client application which enables users to sign-in and issue requests for protected resources. The jsrsasign 8.0.4 Javascript library is used for key and PoP Token generation.</td>
</tr>
<tr>
<td>dv12rwt/pop-angular</td>
<td>A set of microservices including a Token Service with Key Set read-side and a Performance Service used to obtain authorization performance metrics.</td>
</tr>
<tr>
<td>dv12rwt/pop-lagom</td>
<td>An extension of the Nimbus JOSE + JWT library by Connect2id which provides support for Client bound Access Tokens and PoP Tokens.</td>
</tr>
</tbody>
</table>

Table 8: The implementation used to measure PoP Token performance
6 Analysis

The additional protection provided by PoP Tokens comes at a cost. Bearer Token usage, from a Client’s perspective, is as simple as including the Access Token in the Authorization header. PoP Token usage requires Clients to generate a key pair when a user signs in and generate a new PoP Token for each request.

PoP Token usage also requires extra work to be done when authorizing requests. Instead of just verifying and validating an Access Token, as is the case for Bearer Tokens, PoP Token authorization involves an additional signature verification and validation involves computing the S256 digest of the request body.

The implementation was used to quantify the cost of Client key generation, PoP Token generation and PoP Token based authorization for the signing algorithms specified in Section 2.4.1. The results are used to provide a performance comparison between PoP Tokens and Bearer Tokens and draw conclusions about the feasibility of PoP Token based authorization in production environments.

The Client application was tested using a Mobile and a Desktop platform. The microservices were run on a single server. The platform specifications are shown in Appendix A.

The log-transformation [26] was used to yield a symmetrical distribution from the positively skewed computation times. Average times refer to the geometric mean and the reported confidence intervals assume the data follows a log-normal distribution. Tables and graphs are presented in the original scale.

6.1 PoP Key Generation

PoP Key generation is performed by the Client upon receiving an Authorization Code. The PoP Key Generation time is experienced by the end-user as a post authentication delay: a delay between submitting authentication credentials and becoming fully signed in.

The Client implementation uses the jsrsasign 8.0.4 Javascript library [24] to generate key pairs. Average PoP Key generation times are shown in Table 9 and Figures 5 and 6.

The time it takes for a Client to generate a PoP Key varies greatly on the key type and platform. Currently RSA key lengths of less than 2000-bits and ECDSA keys of less than 250-bits are not considered to provide adequate security [21]. The 1024-bit RSA key generation time should therefore be viewed as a reference figure and not as a recommendation of use. All RSA based signatures use a key size of 2048-bits unless stated otherwise.

The results show that Clients were able to generate an EC key-pair in roughly 13% of the time it took to generate an RSA key with a similar level of security. Key generation, however, is only performed when the user signs in. If Access Tokens are issued with a lifetime of 24 hours users will not have to endure this delay more than once a day.
## 6. ANALYSIS

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Generation Time Mobile (ms)</th>
<th>Generation Time Desktop (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA (1024-bit)</td>
<td>1086, (1006, 1173)</td>
<td>124, (115, 135)</td>
</tr>
<tr>
<td>RSA (2048-bit)</td>
<td>12524, (11446, 13705)</td>
<td>1383, (1260, 1518)</td>
</tr>
<tr>
<td>EC (256-bit)</td>
<td>166, (160, 171)</td>
<td>17.0, (16.8, 17.2)</td>
</tr>
<tr>
<td>EC (384-bit)</td>
<td>381, (379, 384)</td>
<td>41.2, (41.1, 41.3)</td>
</tr>
</tbody>
</table>

Table 9: Average Key Generation Times with 95% confidence interval for Mobile and Desktop

![Key Generation Times on Desktop](image)

Figure 5: Average PoP Key generation times on Desktop
The choice of Client key type must also take into consideration the time it takes to generate PoP Tokens.

6.2 PoP Token Generation

Each time the Client issues a request it must generate a PoP Token. The time this takes adds to the response time experienced by the end-user. Average PoP Token generation times for a 1kB request are shown in Table 10 and Figures 7 and 8.

<table>
<thead>
<tr>
<th>Signing Algorithm</th>
<th>Generation Time Mobile (ms)</th>
<th>Generation Time Desktop (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS256</td>
<td>478.6, (473.7, 483.6)</td>
<td>29.8, (29.6, 29.9)</td>
</tr>
<tr>
<td>RS384</td>
<td>498.2, (493.7, 502.7)</td>
<td>30.8, (30.6, 31.1)</td>
</tr>
<tr>
<td>RS512</td>
<td>483.5, (480.6, 486.3)</td>
<td>31.1, (30.9, 31.3)</td>
</tr>
<tr>
<td>ES256</td>
<td>212.0, (208.9, 215.1)</td>
<td>17.2, (17.0, 17.4)</td>
</tr>
<tr>
<td>ES384</td>
<td>513.4, (509.6, 517.1)</td>
<td>42.2, (42.1, 42.3)</td>
</tr>
</tbody>
</table>

Table 10: Average PoP Token Generation Times with 95% confidence interval for Mobile and Desktop

The results show that similar PoP Token generation times can be expected for the RSA based algorithms. The longest generation times were experienced when
using the ES384 algorithm. However, significantly lower generation times were achieved using the ES256 algorithm, even in the Mobile browser. This makes it a good choice of algorithm from the Client’s perspective. The following section will look at the effect the signing algorithm has on authorization times.

Figure 7: Average PoP Token generation times for selected signing algorithms on Mobile

Figure 8: Average PoP Token generation times for selected signing algorithms on Desktop
6.3 Authorization Time

The time it takes services to perform authorization is divided into two sub tasks: Access Token validation and PoP Token validation. Here validation refers to token verification and validation as described in Section 4.5.

Bearer Token based authorization times correspond to the Access Token validation time. PoP Token based authorization times correspond to the sum of the Access Token validation and PoP Token validation times. The results are shown in Figure 9.

![Token Validation Times](image)

Figure 9: Average PoP and Access Token validation times for selected signing algorithms

When an RSA based signing algorithm is used, the overhead Access Token validation has on request processing is extremely low (less than 1ms). This makes RSA based algorithms the best choice for signing Access Tokens.

Clients choosing ES256 over an RSA based algorithm to minimize PoP Token generation times will see an increase in authorization time of about 74% (from 4.11 to 7.14ms in the implementation tested). The advantages for the Client more than outweigh the server-side cost. This means that the EC256 algorithm represents a good choice for PoP Token signing.

Table 11 shows the average times observed for Bearer Token and PoP Token based authorization. PoP token validation proves to be a significant overhead in request authorization. Authorization times are increased by a factor of between 6 and 9 times compared to Bearer Tokens signed with an RSA based algorithm.
7. DISCUSSION

<table>
<thead>
<tr>
<th>Signing Algorithm</th>
<th>Bearer Token Authorization Time (ms)</th>
<th>PoP Token Authorization Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS256</td>
<td>0.71, (0.70, 0.73)</td>
<td>4.23, (4.13, 4.33)</td>
</tr>
<tr>
<td>RS384</td>
<td>0.69, (0.67, 0.71)</td>
<td>4.11, (4.01, 4.21)</td>
</tr>
<tr>
<td>RS512</td>
<td>0.68, (0.67, 0.70)</td>
<td>4.18, (4.10, 4.27)</td>
</tr>
<tr>
<td>ES256</td>
<td>4.59, (4.43, 4.75)</td>
<td>11.04, (10.68, 11.40)</td>
</tr>
<tr>
<td>ES384</td>
<td>9.25, (8.90, 9.61)</td>
<td>17.11, (16.64, 17.59)</td>
</tr>
</tbody>
</table>

Table 11: Average PoP and Access Token validation times with 95% confidence interval

7. Discussion

The primary aim of this project was to address the security issues associated with Bearer Tokens through proof-of-possession. This was done by binding Access Tokens to Clients by including their public key and specifying a method Clients can use to prove possession of the corresponding private key.

The advantages of token based authorization; the ability to make an authorization decision without the need to fetch authorization credentials, is preserved. This is essential for authorization performed by loosely coupled microservices. Services can independently perform token verification, validation and access control. In addition a scalable Token Service is described which can be used to issue Access Tokens.

The increased security PoP Tokens provide can easily be compromised by the protocol used to issue tokens. To address this problem a protocol which uses an external OpenID Connect compliant identity provider is also specified. This can be used to issue any custom Access Token type to any type of Client (native or web-based). This drastically reduces the choices developers must make when using OpenID Connect, reducing the risk of introducing security vulnerabilities through improper OpenID Connect usage.

The final aim of the project was to judge the feasibility of PoP Tokens. To aid this evaluation some basic performance metrics were gathered from a test implementation. However, the feasibility of the token type does not only depend on weighing up the security increase versus performance decrease. The complexity of Client implementations must also be considered.

7.1 Future Work

The performance metrics gathered from the test implementation gave some useful data, however it would be interesting to see how authorization times would be affected by varying the PoP Token content. Protection against token leakage
can be achieved without protecting the request itself. This would simplify client implementation and speed up authorization times.

Authorization times may also decrease significantly if services did not protect against replay attacks as this involves database access. The cost of this operation may also be explored.

The Client application developed for testing was run in a web-browser. A native mobile application may perform significantly better and would provide further data on which to judge the feasibility of PoP Tokens.

The increased authorization times may also leave services more susceptible to denial of service attacks with carefully selected invalid token formats. This should not differ from Bearer Token usage as the Access Token is verified first, however further exploration of this risk warrant further work.

On a related subject, it would be desirable for the system to include token revocation capabilities. Further work in this area is also required.
8. Conclusion

The analysis shows that Elliptical Curve cryptography may be employed by lightweight devices to reduce Client key and token generation times. This comes at the cost of increased authorization times, however the advantages for Client applications outweigh the costs. From the end-user’s perspective PoP Tokens seem viable; the effect on response times is acceptable.

PoP Tokens effectively address the security issues related to Bearer Tokens. Requests can be performed over an insecure channel and are protected against modification and replay. Moreover, PoP Token based authorization can be performed independently, maintaining loose service coupling and scalability. PoP Tokens also present a viable solution in this respect.

Requiring Clients to generate a PoP Token for every request is very different to requiring the simple inclusion of a Bearer Token. The development of Client applications becomes significantly more difficult, especially in the absence of libraries for various platforms to aid PoP Token generation. An accepted PoP Token standard would allow for the development of Client libraries.

The largest obstacle standing in the way of application level proof-of-possession is undoubtedly the increased Client complexity. Current work on binding tokens to TLS connections could provide protection against token leakage without adding complexity to Client applications. However, such techniques rely on the security of the TLS connection and cannot be used in the absence of TLS. For these reasons there are still situations which may require application level proof-of-possession. In this case the methods described here present a viable option for increased security in a microservice environment.
References


[2] Auth0. Identity Providers Supported by Auth0. (Accessed on 12/21/2017). URL: [https://auth0.com/docs/identityproviders](https://auth0.com/docs/identityproviders)


[29] OASIS. SAML V2.0 Holder-of-Key Assertion Profile Version 1.0. URL: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml2-holder-of-key-cs-02.html (visited on 09/20/2017).


[34] Single Sign On & Token Based Authentication - Auth0. (Accessed on 12/19/2017). URL: https://auth0.com/

Appendices

A Testing Platform

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop</td>
<td>7.5 GB</td>
<td>Intel® Core™ i5-4210M CPU @ 2.60GHz × 4</td>
</tr>
<tr>
<td>Lenovo L440, Ubuntu 16.04 LTS, Google Chrome v61.0.3163.100 (64-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mobile</td>
<td>2.0 GB</td>
<td>Qualcomm® Snapdragon™ 800 CPU @ 2.26GHz × 4</td>
</tr>
<tr>
<td>Galaxy S4, Android v5.01, Google Chrome V63.0.3239.111 (32-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>8.0 GB</td>
<td>AMD Athlon™ X4 740 CPU @ 3.2GHz × 4</td>
</tr>
<tr>
<td>Ubuntu 16.04 LTS, Java™ SE Runtime Environment (build 1.8.0_151-b12)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Test platform specifications