Tomiko:

An Extensible OAuth 2.0 and OpenID Connect Authorization Server

Jason Pizzuco

School of Computer Science

McGill University, Montréal

April 2023

A project report submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Computer Science

Contents

Chapter 1

Introduction and Contributions

Tomiko is an extensible OAuth 2.0 [1] and OpenID Connect [2] authorization server and identity provider. Tomiko is compliant with many existing software and protocol standards and specifications in its domain, but seeks to extend the coverage of these standards by creating general solutions to the problems they leave unspecified.

Useful standards tend to be written in ways that make their software implementations useful in many different situations. These standards tend to be limited in scope and fit together with other standards to build a useful system. Following this pattern, Tomiko is designed to decompose the non-standardized "implementation detail" area of authorization servers and identity providers into distinct concerns. To this end, Tomiko has multiple components, each of which communicate together using new interfaces. These interfaces (referred to throughout this report as APIs) represent the different concerns the components are responsible for.

Unlike most existing authorization servers and identity providers which are monolithic and can only be extended by making their codebases even larger, Tomiko is designed so that

its components may be freely replaced or extended with alternate implementations so long as they are able to communicate using the APIs expected by the other components in the system.

In the context of this project, the Tomiko system was designed, reference implementations of components were built to help refine the design, interface definitions, and abstraction boundaries, and those reference components were deployed into production by multiple organizations in order to ensure their functionality.

This report describes the overall space and most important specifications surrounding OAuth 2.0 and OpenID Connect, both of which are implemented by Tomiko. A description is given of Tomiko's overall design and of the different components, what each one is responsible for, and how they communicate together. A sample of 3 organizations that use Tomiko are studied as cases justifying elements of Tomiko's design, outlining the problems it can solve, and the techniques used to solve them. Finally, Tomiko is compared to existing software, its implemented protocols are compared to other protocols used to solve some of the same problems. A brief overview of how its reference components can be hosted and scaled is also presented.

Chapter 2

Background

This chapter describes the OAuth 2.0 and OpenID Connect specifications, their purpose and use, and how Tomiko fits into these specifications. It later describes some of the processed involved in carrying out an authorization according to these specifications.

The first section establishes terminology from the OAuth 2.0 and OpenID Connect specifications that are used throughout the report. The next section helps to build familiarity with the basics of OAuth 2.0 by walking through the most commonly used authorization grant type, the Authorization Code Grant [1].

2.1 OAuth 2.0, OpenID Connect, and Tomiko

This section describes the OAuth 2.0 and OpenID Connect specifications. Specifically, we outline the different entities involved in an authorization and their respective responsibilities toward the end-to-end process.

Tomiko is an extensible OAuth 2.0 and OpenID Connect authorization server and identity

provider. In contrast to some authentication methods, where a system that wants to protect access to a resource must implement authentication and authorization on its own, different entities perform different roles and communicate during an OAuth 2.0 authorization process to acquire the required authorization to access the resource. These roles are as follows.

Resource Owner An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

Resource Server The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Client An application making protected resource requests on behalf of the resource owner and with its authorization. In the OpenID Connect specification, this is also known as a *relying party*.

Authorization Server The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

OAuth 2.0 concerns itself only with authorization, not authentication. The methods used to identify or authenticate the user are out of scope for the standard, so different authorization servers implement that concern differently. Doing so securely, especially in a way that allows the authorization server to be usable in different contexts or with different configurations, is not trivial. This limitation means that the entities performing other roles in the process do not participate in authentication in a standard way and must be prepared to adapt to custom processes.

OpenID Connect [2] is an extension to OAuth 2.0 that adds additional features, but also adds additional constraints to implementers to simplify interoperability. OpenID Connect identity providers (i.e., authorization servers that are compliant with the OpenID Connect extension) are also usable to provide a means for standardized authentication alongside authorization.

These standards define a minimal protocol for relying parties to begin the authorization or authentication process, for authorization servers to communicate the termination of the process and result to the requesting relying party, and for providing assertions about the completed process to relying parties that they can present to resource servers in order to access resources.

Many important aspects of the process are left as implementation details. Implementers must answer design questions like: how the process will interact with other systems and components once it has begun; how the resource owner provides access, credentials, or consent; and how resource servers can validate authentication or authorization assertions.

Since identity providers are somewhat complex, most are purpose-built for a particular organization, social network, or service provider. These are usually very large monolithic systems whose components (to the extent they're even separated into separate components at all) are not reusable to build other providers.

Tomiko provides a fully-functional implementation of these specifications that is designed to be extensible and reusable, and whose internal components interact using a well-defined protocol so that they can be replaced with different implementations if necessary without having to redesign the entire system.

2.2 Authorization Process

This section describes the OAuth 2.0 Authorization Code Grant. Most applications using OAuth 2.0 and/or OpenID Connect use this or a variation of this grant. It builds off the previous section by describing which parts of the grant each role bearer is responsible for.

When a resource owner is using a client and would like to authorize the client to receive a protected resource from a resource server, the client makes a request to the authorization server to ask for access to the resource.

```
{
  "response_type": "code",
  "client_id": "my-app-id",
  "redirect_uri": "https://my-app.my-org.com/callback",
  "scope": "my-app-scope1 my-app-scope2",
  "state": "my-app-opaque-string"
}
```

Figure 2.1: A simple authorization request

A standard OAuth 2.0 (potentially with OpenID Connect extension) authorization request is shown in Figure 2.1. It takes the form of a HTTP `GET` request to *tomiko*'s authorization endpoint with the following parameters encoded into the query string (parameter values containing the substring `my-app` are placeholders that will be filled by the client). The `response_type` parameter describes the type of grant the client wishes to use. In this example, we will describe an *authorization code* grant, so the parameter value is `code`, but

the specifications define multiple different grant types which have entirely different authentication flows. All grant types specified in the OAuth 2.0 and OpenID Connect specification (as well as other extension grant types from other specifications) are supported by Tomiko. The `client_id` parameter is an identifier assigned to the client by the authorization server. The `redirect_uri` is a URI to which the caller will be redirected after authorization has taken place. For security reasons, this URI must be whitelisted and associated with the requesting client to ensure that malicious or compromised applications do not build authorization code relays to untrusted endpoints. The `scope` parameter is a space-delimited list of permissions that the client is requesting that the access token cover. For example, a photo sharing application may request a scope to get access to a user's contacts/friends and a user's photo library. This list is later used to display a consent screen to the user to make them aware of what permissions they are authorizing the client to have on their behalf. The `state` parameter is an opaque string carried through the request and and sent-back to the client alongside the redirect response. The client can use this to associate information with the authorization request (such as the action the user was trying to perform when they were asked for authorization or authentication, so that they can continue where they left off once the process completes successfully).

At this point something interesting must happen: the user (currently within the client application) must be directed to the authorization server so that the authorization server may interact with the user and establish their authorization. In order to do this, the authorization request must happen within the user's user agent. This places an interesting constraint on the content of the authorization request: it cannot contain any secret data that the authorization server might use to authorize the client, and its response cannot contain data that cannot

be assumed to be intercepted by clients other than the one making the request.

Somehow, in an implementation-dependent way, the user who has now been directed to the authorization server is authenticated with the server and their authorization to grant the client application access to the requested resources is established.

The authorization server now generates an authorization code and sends it back to the client using the `redirect_uri` from the original request. An example response can be seen in Figure 2.2 (note that values containing `my-idp` are placeholders chosen by the authorization server and values containing `my-app` are placeholders chosen by the client). This authorization code is a one-time-use secret that can be exchanged for an access token asserting the user's authorization upon the client's successful authorization.

The client, having received the authorization code, must now make a second request to the authorization server in order to receive an access token. Since this response to this request will contain a token which the client must ensure does not leak, the client application makes this request internally (`not` using the user's user agent).

A sample request is shown in Figure 2.3. In this request, the grant type for this request is specified to be `authorization_code`. The authorization code is transmitted in the `code` parameter, and the URI the user was redirected to in order to capture the code is transmitted in the `redirect_uri` parameter. Finally, since acquiring a token is a privileged action, the client provides its own authentication in the `client_id` and `client_secret` parameters to prove that the authorization code wasn't stolen in transit but was actually issued to this same client.

An example response to this request which is shown in Figure 2.4. The opaque access token is transmitted in the `access_token` field, the token type (usually used when specifying

the authorization scheme in the HTTP Authorization header) is specified in the `token_type` field, and number of seconds for which the token is valid is transmitted in the `expires_in` field.

```
{
  "code": "my-idp-code",
  "state": "my-app-opaque-string"
}
```

Figure 2.2: A simple authorization response

```
{
  "grant_type": "authorization_code",
  "code": "my-idp-code",
  "redirect_uri": "https://my-app.my-org.com/callback",
  "client_id": "my-app-id",
  "client_secret": "my-app-secret"
}
```

Figure 2.3: A token request

```
{
  "access_token": "my-idp-token",
  "token_type": "my-idp-token-type",
  "expires_in": 3600,
}
```

Figure 2.4: A token response

Chapter 3

Design and Methodology

This chapter describes Tomiko's design and how is components are laid out and communicate together. Each of its first sections describe a single component and its responsibility toward the whole, as well as which other components it communicates with in order to implement its concerns. Finally, a description of the overall architectural design and extension patterns is provided to convey why Tomiko is designed the way it is.

A Tomiko deployment is composed of 3 or 4 separate components depending on how it is extended. Tomiko, while naming the entire system, is also the name of the core component. When referring to the component specifically, it will be typeset in italics.

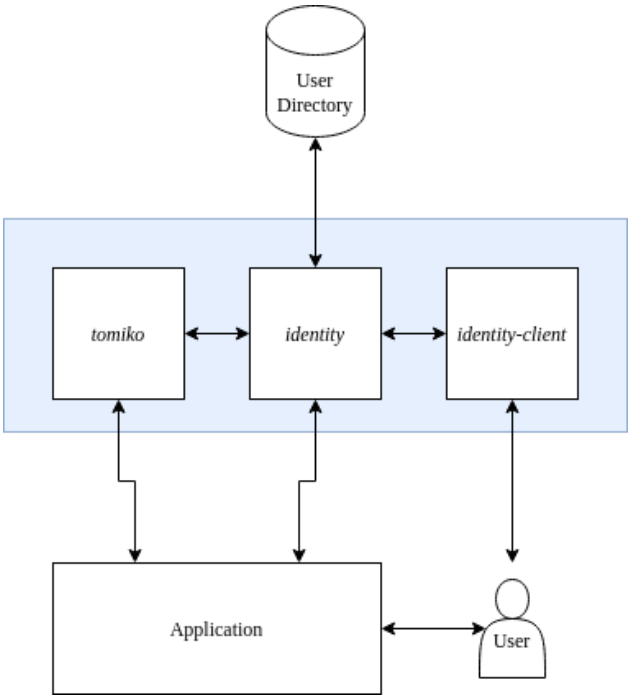

Figure 3.1: Architecture block diagram

3.1 Overall Architecture

Tomiko's overall architecture is designed to be extensible by defining and stabilizing the APIs components use to interact with one another. Rather than being a large monolithic system that is extended through framework-style built-in support for different functionality or through configuration, each of Tomiko's components have restricted scopes and encapsulate few different concerns as possible. This allows components to be *replaced* with equivalent ones that better meet a user's needs.

This enable this, components implement one or more HTTP-based APIs. Tomiko is designed to assign each of these APIs a single responsibility/concern so that components do not have to rely on knowledge of which software implements unrelated APIs or how that software implements them.

In a standard Tomiko deployment, there is minimal cross-talk between components - each component usually communicates with one other component to implement a concern. A block diagram indicating which components communicate together is presented in Figure 3.1. Applications relying on Tomiko's services communicate with *tomiko* over a standard OAuth 2.0 interface with standard OpenID Connect extensions. *tomiko* instructs the relying application with where it should direct the user to complete the authorization the application is requesting. Since the OAuth 2.0 protocol standard specifically states that the mechanisms used to authenticate a user or establish their authorization are out of scope for the standard, *tomiko* mirrors this extension point and delegates all control over authentication and authorization away, concerning itself only with presenting and processing assertions about the authentication or authorization process after it as taken place. This means the user, while performing their portion of the authorization process, does not have any need to interact with the *tomiko* component. The user may need to present some tokens in order to authenticate themselves at some frontend interface that a human can interact with, so *tomiko* instructs the application to direct the user to *identity-client*.

It's worth noting that *tomiko* has no particular coupling to *identity-client* or its implementation. Rather, *tomiko* encodes information about the request (in this case, a *challenge identifier*) that any component that can interact with its challenge API can use. *tomiko* is simply configured with a URI a requestor can use to reach a component that can perform any authentication or authorization validation requried, and encodes the challenge identifier into the URI's query string. Where this URI points or what is done with the challenge identifier is out of scope for *tomiko*. Likewise, for *identity-client*, when a request containing a challenge identifier is received, it does not necessarily need to know which identity provider

issued the challenge. Depending on how Tomiko is extended, the frontend component may be implemented with the ability to perform authentication, authorization, or both on its own. Whichever of these tasks it does not implement can be delegated to another component. In either case, *tomiko* does not assume or have mechanism to enforce any particular behaviour.

In the standard case, *identity-client* interacts with the user to request tokens or consent, but does not have any mechanism to validate them or decide which tokens are required or in which order. *identity-client* also does not have any interpretation of any of the scopes requested in an authorization request. Instead, *identity-client* is driven by APIs presented by *identity*.

identity is responsible for identifying and authenticating the user, and any mechanisms, directories, or requirements for doing so are configured only within *identity*. When a request arrives at *identity-client*, this component asks *identity* how to proceed. *identity* drives a state machine to determine which authentication tokens/factors are required, and communicates each of these to *identity-client* in sequence. For example, when *identity-client* requests an indication on what to do next, *identity* might reply signaling that a `password` factor is required, and provides any information that may be required to ask for a password (such as the user's name to show them a greeting). *identity-client* receives this reply and displays a form asking the user for a password. The user sees this form and provides a password, which *identity-client* sends back to *identity* asking "here's the password you asked for. what next?". *identity* may repeat this process by replying with requests for any number or sequence of factors until it is satisfied with the user's authentication. This process is illustrated in Figure 3.2

Once it is satisfied, *identity* may reply to *identity-client* that it's time to show a consent

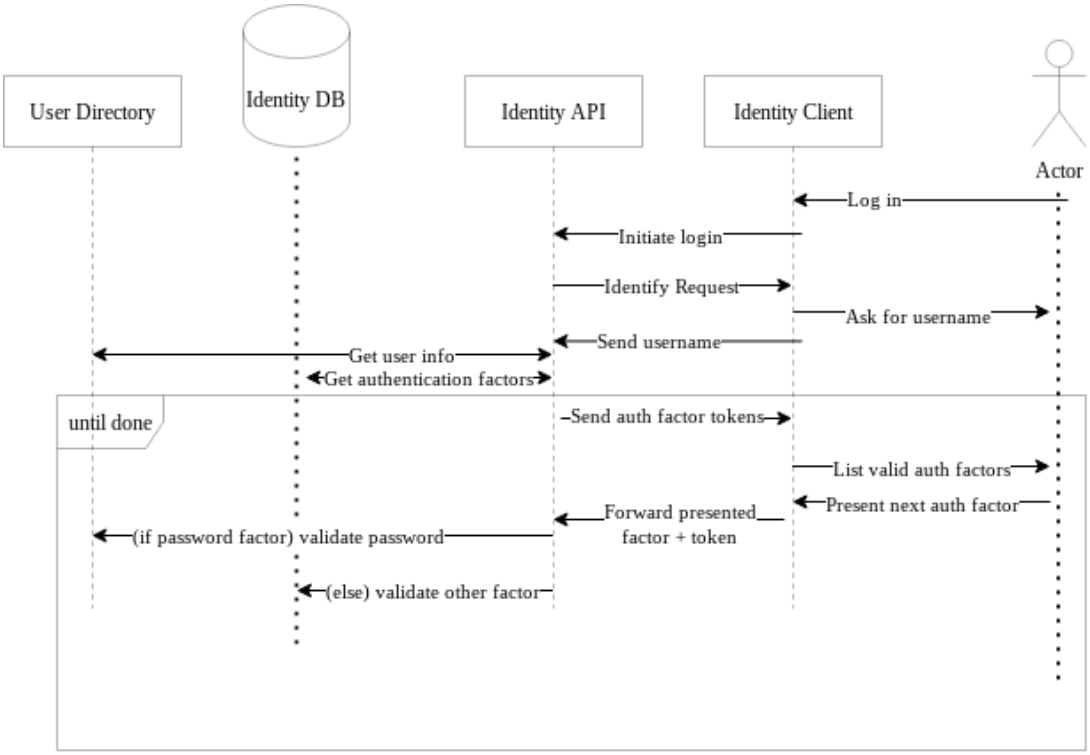

Figure 3.2: Identity login flow

screen, and provides the information (including which scopes are being requested).

The entire interaction, beginning with a user visiting an application is summarized in Figure 3.3

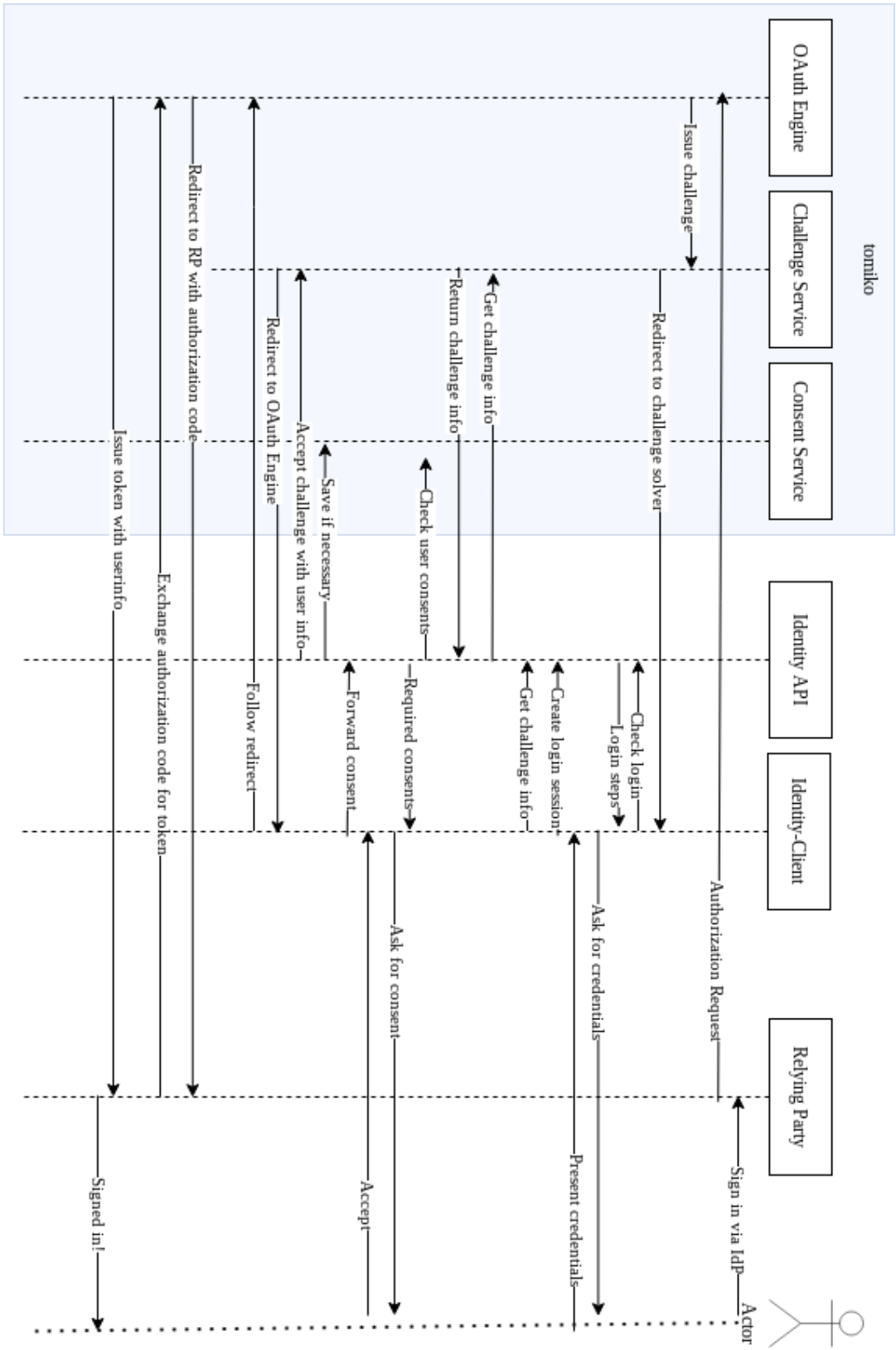

Figure 3.3: Tomiko Authorization Code Flow

3.2 Tomiko

tomiko is the core component that implements the authorization server interface. Relying parties interact with Tomiko by exchanging messages to *tomiko* almost exclusively. *tomiko* aims to follow the design of the OAuth 2.0 standard [1] very closely - it is not concerned with the answers to most of the questions implementers must answer. Instead, *tomiko* implements the core OAuth 2.0 protocol in a generic but extensible way, and provides an API to interact with separate and interchangeable components that handle the usually implementation-specific portions of the authentication or authorization process.

tomiko can authenticate relying parties and resource servers using standard OAuth 2.0 methods, but cannot itself authenticate users. *tomiko* tracks which authorization or authentication processes are currently active and what each of their states are, and provides other components in the system with information to perform their role. When the process is complete, it provides a response to the original requester, in a format that can later be validated by systems that would like to.

Of course, once an authentication or authorization process has begun, the directory against which to validate any user credentials must be known, and this will vary based on the user of the software. While other authorization servers have a list of supported user directories (most of the time only one option), *tomiko* relies upon an external component interacting with its challenge API. In a standard Tomiko deployment, the component that interacts with this API is called *identity*.

3.3 Identity

identity is the component which performs authentication decisions, checks user directories and login policies, handles multi-factor authentication mechanisms, and responds to *tomiko* challenges through its API. *identity* supports different user directories, and each are fully configurable. In particular it supports LDAP directories, which are the most common in organizations seeking centralized authentication. While some other IdP LDAP plugins only support a short list of LDAP schemas (usually only Active Directory-style schemas), *identity* has an entirely configurable schema mapping. *identity* can either be configured through simple configuration files or environment variables, or custom plugins can be written in Ruby to arbitrarily extend the functionality in ways that might be more difficult with declarative configuration.

identity interacts with *tomiko*'s challenge API and any external services needed to validate the authenticating user against the configured directory or multi-factor authentication systems. In order to actually interact with the user, *identity* directs the user-agent to a web-based frontend, *identity-client*.

3.4 Identity-client

identity-client interacts with *identity*'s APIs, meaning it can also be switched out for any other frontend that can interact with the user on *identity*'s behalf. The standard *identity-client* is a dynamic in-browser single-page application that presents a rich user interface to the user. Since different organizations might desire a different "look and feel" or want to present different information to users with particular organization-defined roles, *identity-*

client is also configurable so that it does not have to be replaced for most use cases. It provides two options for extensibility, directing users to a separate component hosted at the same domain for a list of views, or writing custom components that are hosted within the same single-page application. If the latter option is chosen, these components can interact with the rest of *identity-client* though well-defined APIs with TypeScript annotations.

Chapter 4

Validations

Tomiko was origininally designed to meet the needs of a particular organization, and its architecture was originally conceived to address that organization's needs only. Initially developed as a proof of concept, as it came into use and showed evidence that it could address the problems it set out to solve, there was increased interest in adapting Tomiko to work in other organizations – some of which had vastly different needs or surrounding infrastructure to interoperate with.

Tomiko is now used in production by a variety of different organizations. With these groups, it has been proven to work well with different OAuth 2.0 and OpenID Connect components.

It has been integrated with commercial, free/open source, and custom relying parties. Since OAuth 2.0 and OpenID Connect are both open standards that are designed to allow compliant software to interoperate with each other, Tomiko's ability to interoperate with out-of-the-box components is an initial testament to its correctness.

Of course, these protocols seem deceptively simple, so blackbox testing at the edges of

the interface would not be sufficient as a single validation to show that Tomiko properly addresses what it sets out to.

One of Tomiko's main selling points is its extensibility model. This is precisely what allows Tomiko to go beyond an organization-specific in-house identity provider to an extensible platform for building compatible identity provider components.

To validate whether Tomiko's extensibility points are sufficient to allow it to be used in different ecosystems, we will analyze some case studies of different organizations using Tomiko, and how their needs helped to shape Tomiko's design.

This chapter describes how Tomiko came to be, and gives case studies that helped to shape its design and help to validate that it meets its design goals. Each of its first chapters is a case study describing an organization with specific requirements for any identity provider they might come to use, explains how Tomiko met or was extended to meet those requirements, and how that organization came to use Tomiko. Each organization is pseudonymous.

4.1 Organization A (a large student organization)

The first organization to use Tomiko, which we'll refer to as Organization A, was a student organization at McGill University. This is an organization composed primarily of student developers, and it makes heavy use of free/open source tools.

Control over management interface Since most of its members are highly technically competent and would like to build many different applications that use Tomiko as an authentication and authorization provider, the ease with which new integrations could be added

and with which information was available about each user in the system was an important factor.

In addition to this, the organization had a mature governance structure surrounding which roles should have access to change information within the system (not just anybody could self-enroll relying parties or change their properties, it had to be done centrally through an approval process).

The organizational governance structure surrounding changing information or adding integrations to the system actually made the implementation simpler. In highly dynamic environments (such as social networks that do not manually verify every third-party application that wants to allow users to authenticate through them), authorization servers must usually support a specification called OpenID Connect Dynamic Client Registration [3]. This is a complex specification that allows applications to dynamically request new client registrations/credentials and modify information about those clients. This can sometimes be convenient, but is not useful for most organizations and can widen the security perimeter unnecessarily. Not having to implement this standard simplied Tomiko's requirements and necessitated coming up with a different way to enroll and manage clients.

To do this, a command-line management interface which would be friendly to automation tools/pipelines was developed which allowed client enrollment and management. This meant organizations could build their own governance and processes surrounding these actions and easily perform the actions either manually or in automated workflows when appropriate.

User Attributes This organization delegated its user directory to McGill, so it did not have to own user information or handle user provisioning. The university used Microsoft

Active Directory to hold user information. Before Tomiko, Organization A handled authentication by configuring each of their software individually to interact with Active Directory through LDAP [12]. Since Organization A did not own the data in Active Directory, it did not have control over the schema or format of the attributes. As a result, some of the attribute values required filtering or processing before they were meaningfully usable by applications that were interested in them. Some attributes were not directly present in Active Directory, but were instead derived by projecting a combination of other attributes into a new "virtual" attribute.

Tomiko was replacing an older attempt at solving the problem of delegating LDAP authentication. That older system handled both LDAP integration and token generation, but this proved to be fragile to maintain because it required understanding both these concerns to make any changes to the system. It also prevented the parts from being independently reusable. That system had two components, one which handled LDAP authentication and token generation, and another which exchanged valid tokens for information about the user that authenticated (which also came from LDAP). Since the LDAP portion was embedded into the first component, it was not easy to reuse it in the second component, which also needed to perform LDAP operations. Likewise, information about how the token was generated in the first component was necessary to properly validate it in the second component, but because the concerns were combined with the authentication logic, it was difficult to reuse them.

This led to the decision to separate all Tomiko's token management from its user management. Not only did this more closely follow the OAuth 2.0 specification's design (since user management was not in scope of that specification), it greatly simplified and centralized the

logic for each of these concerns. Thus, the *tomiko* component would handle OAuth 2.0 tokens and a second component, *identity* would handle user management. This meant that *tomiko* could be completely independent of LDAP, so it could be used by organizations that backed their user directory by different systems. Likewise, *identity* could be completely independent of OAuth 2.0. It was built to custom abstractions that mirrored OAuth 2.0's abstractions (clients and scopes) but that could be backed by other protocols if needed later, and did not require knowledge of the implementations of those abstractions to perform maintenance on the user management. This meant that student developers who required new user management features could feasibly add them to *identity* without any particularly deep knowledge of how OAuth 2.0 works and without any fear of introducing vulnerabilities into the token generation process.

Initially, *identity* was purpose-designed to be used with McGill's Active Directory layout, but the requirement to support "virtual" attributes meant that it would not be sufficient to naively map known attributes to user information claims. Some of these attributes required processing, and there may be cases where one claim comes from multiple attributes, or a subset of the values for a single attribute. Since we had many attributes to manage, and developers would probably need to add or modify these attribute definitions, special care was taken to ensure that it would be simple to do so without having to know too much about the surrounding system. To support this, a system whereby information claims could be specified using configuration files rather than modifying the source code was developed. This system needed to allow all claims (including the base set defined in the OpenID Connect Core specification[2]) to be defined and mapped to LDAP attributes, but also required relatively complex operations and projections of custom attributes. This configuration file

layout was inspired by declarative YAML CI/CD pipeline definitions, and allowed defining attributes that came from multiple configurable sources, as well as pipelines of regex and other transformations to derive the final value. This format proved over time to be easy to use and extend.

Responsible data use Many factors led to concerns about responsible data use within this organization. First, user data came from the university and could in some cases be considered to be sensitive. Second, Organization A offered a platform for external student developers to build applications using the organization's APIs and integrations, and there were concerns about how to ensure that those external developers used user information responsibly and made users aware of which data was collected.

While generally Organization A preferred that all user information required that user's explicit authorization before being disclosed to an application, there were exceptional cases where some information needed to be available even if the user was not present or had not yet had the opportunity to authorize the disclosure. For example, organization members who performed user support tasks might need to look at information about a user who is having trouble authenticating (and therefore could not have performed any authorization) to see whether there was a problem with the user's profile.

Concerns over responsible data use led to special emphasis on authorization of user information release. It was important to design a system that allowed uses to understand which information was being required of them, who was requesting it, and decide whether they wanted to authorize it. These pieces of data are linked to OAuth 2.0 scopes. If an application requires a user's email address, it can request the "email" scope as part of the

authorization request. Handling these scopes is tricky, because it must necessarily involve multiple components. Since scopes are requested in the authorization request, which goes to *tomiko*, that component must be aware of scopes. Since *tomiko* will eventually issue the token if appropriate, and the token contains the list of authorized scopes, it must ensure that the list of scopes in the token is really the same list the user actually authorized. However, *tomiko* does not interact with the user in any way, so it has no way to directly inform or ask the user to authorize. Instead, *identity-client*, the only component that directly interacts with the user must accurately display the scopes and capture the user's authorization assertion. However, *identity-client* does not speak directly to *tomiko*, only to *identity*. This raises a question about which component owns the responsibility to ensure the scopes are well-handled. Tomiko's solution to this problem is for *identity* to own this responsibility. The *identity* component is configured with the definition of each regular scope a user can authorize. This definition includes any criteria that may disallow a user from authorizing a particular information release (for example, they cannot authorize information relating to services they have not paid for), the name of the scope, and a user-facing description of the scope that users will see when the scope is requested. When a regular scope is present in an authorization request, *tomiko* treats the requested scopes as an opaque set of strings and makes them available in the authorization request context that *identity* can consume later. When a user is directed to *identity-client* to authorize the requested information, *identity-client* acquires the information about the request from *identity*. In turn, *identity* gets the information from *tomiko*'s request context and validates and processes it according to the scope definitions it owns. After trimming away scopes that the user cannot authorize, the description of each scope is pulled from the definition and forwarded to *identity-client* for user interactions. Once

the authorization decision is made, the set of authorized scopes is eventually sent back to *tomiko*, which after performing some security trimming (checking that the final set of scopes is a subset of the original requested scopes to prevent an escalation attack) treats each value in the set of scopes as opaque again when issuing the token. This architecture ensures that scope definitions can easily be modified in a single place, with a single component owning the responsibility to manage them.

In the previous paragraph, the qualifier "regular" was used to describe some scopes. This organization required that in some special cases, some user information could be available to an application even if a user was not present or did/could not consent. To support this behaviour, a separate set of scope definitions is present. These are called "privileged scopes", and permission to use a privileged scope must be explicitly granted to a client. A client must request these scopes as itself, not request them from a user. For example, Organization A defines a privileged scope `priv::all_users:ro`, which allows read-only access to all information about any user. This scope cannot be requested using the regular kind of authorization request that has been described so far, because that process is for allowing a user to delegate its ability to access resources relating to a certain scope to a client. At the end of that process, an access token whose subject is the user who authorized is generated.

In the case of a privileged scope, the semantics are different. An authorization request is not asking a user to delegate a permission to the client, but rather asking the authorization server to provide an assertion that the client itself inherently has a permission. In OAuth 2.0, the standard "client credentials" grant type is used for these kinds of authorization requests, and therefore in Tomiko privileged scopes are only available during "client credentials" grants.

The set of privileged scopes a client is entitled to is part of the client's metadata in *tomiko*, and since no human user is required to authorize them, descriptions of these scopes are not necessary. Since, while the notion of privileged scopes is an abstraction provided by Tomiko and not by OAuth 2.0, but the process to issue them is covered by the OAuth 2.0 standard and does not require external interaction, the responsibility to manage these scopes belongs to *tomiko*, not *identity*. It would not be possible for *identity* to manage them because *identity* is never involved in client credentials grants, since those grants do not involve users. When a client requests a privileged scope during an authorization request to *tomiko*, its permission to use that scope is checked but the scope itself is opaque and carries no inherent meaning. If all the requested privileged scopes are allowed for the client, they will be present in the authorization token that will be generated with the client itself as its subject.

This allows Organization A's example use case of having some applications see user information even if they are not authorized. If there is a support application to look up user profiles to check for problems, that application would ask for a token to be issued to assert that it is entitled to the `priv::all_users:ro` scope. A resource server acting as a user information service which normally only grants access to a user's information if an access token with that user as a subject covering the resource is presented alongside the request could offer the ability to bypass this check and release information about any user if the request is being made by a client which presents an access token asserting that they are entitled to the `priv::all_users:ro` scope.

LDAP software support and bind credentials The organization's existing use of Active Directory through LDAP imposed a hard requirement that any software built or deployed

(commercial, free/open source, custom, or otherwise) supported LDAP authentication. Since LDAP integration is almost always used by large organizations, lots of free/open source software that also offer additional (usually paid) proprietary features do not provide LDAP support gratis or under the usual free license the rest of the software is available under. On the other hand, OAuth 2.0 and OpenID Connect are used for social integrations (such as with Microsoft, Google, Twitter, GitHub, GitLab, etc.) which are very commonly used even in small software deployments, not exclusively used by large commercial organizations, so these features are generally available both free of charge and also under the same free license as the rest of the software.

A minimal LDAP integration was not sufficient, since there was a need to have specific bind credentials and user filters. In order to handle all attributes, the software also required the ability to re-process or re-format data it received via LDAP before using it in a user's profile.

Some user information was available to this student organization that is not generally available to all users at the university, so a special LDAP binding account was used in all of these systems to retrieve access to this information rather than binding to Active Directory using a potential user's credentials as they present them for authentication.

As a result, each application had to be configured with global sensitive bind credentials and had to be strongly coupled to the object schema in Active Directory, and each had to separately implement any logic for processing or deriving "virtual" attributes from the LDAP information. Any security vulnerability in any of these applications that could allow the bind credentials to leak would immediately grant unregulated access to all student information present in Active Directory that that bind account had access to, which includes

user information that was not generally otherwise available. As a result, bind credentials had to be frequently rotated "just in case", which required changing the credentials in every system's configuration separately.

Tomiko's use of OAuth 2.0 and OpenID Connect as identity federation protocols rather than LDAP, and its *identity* component's flexible and centralized configuration of attribute mappings fully addresses these problems. Furthermore, since *identity* is the only component that requires any access to the user directory, it is the only component that ever needs access to the organization's sensitive LDAP bind credentials.

Login restrictions and authentication factors Due to the difficulty of retrieving *user* authorization when using an LDAP-based authentication system, only a very limited form of authorization was used – the organization's authorization for a user to access a particular system.

To validate this, LDAP groups were created within Active Directory for systems which needed this sort of authorization, and users who were authorized to use systems were put into these groups.

At authentication time, after binding to LDAP with the privileged account, the LDAP user whose authentication credentials to test against against those the human user supplied would be found by restricting the search to these authorization groups. If a user was not in the authorization group for that system, they could not be authenticated. The existing system had many problems supporting these use cases.

Second, using base LDAP in this way does not generally allow for multi-factor authentication, since there is no agreed-upon way for all software to determine and process non-

password authentication tokens, which the organization was interested in implementing.

These problems were some of the motivating factors for building Tomiko, so care was taken to make sure they could all be addressed with its design. To address the need that the organization be able to control which users could perform which interactions on which resources across systems, the OAuth 2.0 "scopes" concept was used. OAuth 2.0 clients are already required to specify in their authorization request the list of scopes they'd like to be granted through an eventual access token. While some of these scopes could be *regular* scopes (as mentioned earlier) that allow users to decide whether they would like the client to have access on their behalf, some scopes could also be automatically derived based on user attributes or group membership.

For example, Organization A offers some services with quota restrictions whereby each "use" of the service by a user will consume some of the user's allocated quota to use a functionality of that service. Since all software eventually has issues, some of the organization's members had special permissions to investigate issues with the service and "refund" user quota if there was an error somewhere. When the software that allows management of quota asks a user to log in, the software could request in its authorization request that it would like an access token with a scope that covers the ability to refund quota. When this authorization request is processed, that scope entry can be automatically derived based on any combination of user profile attributes (are they a member of Organization A? are they part of the group of production operators of this service?).

The OAuth 2.0 specification specifically allows for the authorization server to issue an access token with only a subset of the requested scopes, so OAuth 2.0 compliant clients should already be equipped to handle the case that the scope they asked for (in this case, the ability

to refund quota) is not present (either because the user didn't authorize it in *identity-client* or because it was automatically established by *identity* that the user themselves does not have this permission and therefore cannot delegate it) and handle it meaningfully within the application.

To support use of non-password authentication factors, an API was defined between *identity* and *identity-client* to flexibly support prompting for custom authentication factors. As part of *identity*'s authorization state machine, there is a loop over all the configured and eligible authentication factors for the user, and *identity-client* is configured to handle each iteration of this loop by asking for details of the factor and interact with the user to collect the required information.

In Organization A's deployment, *identity* is configured to sometimes require an email confirmation code to be entered when logging in.

4.2 Organization B (a decentralized internet community)

Another organization using Tomiko is Organization B. This is a decentralized internet community of hobbyists, developers, and collaborators that share many self-hosted internet services. Among these services are collaboration platforms, chat platforms, file sharing platforms, federated social software, a source code repository, and a few other pieces of custom and out-of-the-box software. Some members have shared access to the infrastructure that the services are hosted on. Some of this access is via SSH, and other access is through a

secrets broker that requires members to identify themselves before issuing temporary secrets or tokens.

Organization B was the second major user of Tomiko, and supporting it meant updating parts of Tomiko to make it more closely match its goal of being usable and easy to adapt. Before Organization B, some pages inside `identity-client` were hard-coded to make some assumptions about the kind of uniqueness constraints that existed between attributes on a user's profile. The LDAP queries used to query profile information in the LDAP connector also used Active Directory-specific extensions (such as `1.2.840.113556.1.4.1941`, `LDAP_MATCHING_RULE_IN_CHAIN`). Organization B also had an existing LDAP server which contained its users, but used OpenLDAP rather than Active Directory. These changes led to Tomiko having a completely configurable set of LDAP mappings and queries in `identity`.

Since Organization B managed its own users (unlike Organization A), more functionality was needed for users to manage their profile details, such as changing passwords or display names. These were easily integrated into `identity` and `identity-client` by adding the ability to specify "mutable" attributes in the user directory specification and validators for those attributes. An option was added to allow the ability to change passwords for LDAP-backed users on the authentication factors page, which meant refactoring the portion of the authentication stack that previous treated password factors as a special case.

For more complex modifications (such as changing profile photos), `identity` exposed an API to send arbitrary base64-encoded bytes, which meant `identity-client` (or any custom client components mounted by `identity-client`) could perform custom logic to encode any data the user uploaded (such as: in this case, profile photos) before saving them to the user directory.

Organization B offered its members the ability to associate SSH keys with their user profile so that members with infrastructure access could be granted access via their keys. Some members thought it would be interesting to use ownership of their SSH private key as an authentication factor. Since `identity` only supported a single non-password authentication factor at the time (TOTP codes sent via email), implementing an SSH-based authentication factor came to be a way to measure the flexibility of `identity`'s API.

Implementing this required a simple extension to `identity` (the ability to read authentication factor data from the user's profile) and for a microservice to be written implementing the authentication factor's logic which interacts with `identity`'s API. This microservice (called SSHAFT, `SSH authentication factor`) [10] was written in Go (any language could have been chosen, but Go was chosen as a testament to the flexibility of writing extensions in languages no other part of Tomiko is written in) and required only 200 lines of code to be in a production-usable state.

When SSHAFT is enabled and an SSH key is provisioned for a user, the user will have the option to use SSHAFT as a secondary authentication factor after providing their primary factors (username or email and password). If the user chooses this option, a unique identifier is generated for the challenge, the challenge is recorded in `identity`'s database containing a hash of the generated identifier and the user is directed to use their SSH client to attempt to create a connection for `<identifier>@<SSHAFT hostname>`. At this point, `identity-client` would use a WebSocket to wait for new events from `identity` about the status of the challenge.

Meanwhile, the user attempts to create the requested SSH connection. OpenSSH allows a custom command to be run to generate the `authorized_keys` file and passes information

about the connection attempt to this command. The command in this case, is an invocation of *SSHAFT* which connects to `identity` to get the challenge information, tests the user's key against the ones in the user's profile, and instructs `identity` to either accept or reject the challenge. When the result is known, the status is sent to `identity-client` over the open WebSocket to continue or abort the authentication process.

An interesting property of the API between `identity` and authentication factors is that this API requires authorization using OAuth 2.0. The communication between different components uses the same authentication and authorization protocols that the entire system works to provide.

Organization B also provides an email service to its members, authentication to which was traditionally restricted to plaintext passwords via plaintext IMAP and SMTP authentication. The group used to connect their mail servers directly to their LDAP server to verify users' passwords when needed. Since the group was moving to OAuth 2.0 + OpenID Connect based authentication and authorization instead, their mail server and their members' clients had to be reconfigured for the new protocol. Luckily, the mail server software they were using (Postfix and Dovecot, with Postfix delegating authentication to Dovecot via SASL) supported the XOAUTH2 and OAUTHBEARER SASL mechanisms, which allow clients to acquire their own tokens with the required permissions via standard OAuth 2.0 grants and supply the resulting token when connecting to IMAP or SMTP service. Integrating their instance of Tomiko with Dovecot only required a small configuration change to their Dovecot instance. Unfortunately, client support for XOAUTH2 and OAUTHBEARER is not perfect yet. Since clients must acquire their own tokens, they need a way to get client IDs and client secrets and register redirect URIs with the authorization server. Some major clients

do not support custom OAuth 2.0 providers but come with built-in client IDs and client secrets corresponding to registrations with major mail providers (such as Gmail, Outlook, etc). Mozilla Thunderbird is an example of such a client [11]. To try to make support client support as wide as possible, Tomiko supports OpenID Connect Dynamic Registration [3] for a few mail-related scopes, so that clients also supporting this standard can automatically register themselves when the user tries to connect the application to their mail account.

The entire process to customize Tomiko to allow custom LDAP schemas, extend the authentication factor API to allow different factor data sources, and even `build` SSHAFT took one developer less than one week. Using Tomiko with all the software Organization B wanted required no modifications due to Tomiko's standard-compliant implementation of OAuth 2.0 and OpenID Connect.

4.3 Organization C (a centralized community of friends)

Organization C is a much smaller organization than the other 2, having only about 15 users. Its contribution to Tomiko's design is helping to ensure that Tomiko remains very easy to host and configure. While the other two organizations had robust containerized hosting infrastructure, Organization C wanted to run Tomiko on simple multi-purpose shared servers without any container runtime, orchestration, or deployment tool.

This meant Tomiko could not be distributed exclusively as OCI container images, and that the environment setup should be simplified to the extent possible. Previously Tomiko required a setup script that would be run when the container started up to run some sanity-checks and prepare a directory structure before starting the components.

With feedback from Organization C, Tomiko's reference components' startup sequences were modified to bootstrap themselves in a single step on first startup, and multiple configuration files were coalesced into a single configuration file with annotated sections for values that should be changed to customize the components' functionality. Where possible configuration values are read from environment variables rather than configuration files to simplify configuration even further and protect sensitive values.

Organization C also did not use an LDAP directory to house its user information. Users are managed through a manual process and their metadata is read from a text file at startup. This required configuration in `identity`, but did not require modifying the software since reading JSON data and mapping it to attributes was already supported by `identity`.

Members of Organization C all had hardware authentication devices and wanted to use them as secondary authentication factors. With their existing setup based on text files and custom plugins for each piece of software that integrated with authentication or authorization services, this would have been completely impractical. With Tomiko, an authentication factor microservice using already existing APIs in `identity` was built to support hardware authentication devices via the WebAuthn standard[13] (written in Rust in less than 300 lines). `identity-client` was configured to mount a custom JavaScript user interface component to handle the user interaction and the tokens were ready to be used across all their services.

Aside from this, Tomiko was able to be configured and used by Organization C without modifications, and integrated with all the software the organization required.

4.4 Summary

As Tomiko was developed and came to be used by organizations A, B, and C, among others, various aspects of its design were decided or improved upon.

The main contributions toward Tomiko's design linked to Organization A are its management API and tooling, its decoupled user-management and token-management components, its LDAP-based user directory functionality and the configurable attribute and virtual attribute processing pipelines that facilitate its use, its ability to collect and aggregate user information to define OpenID Connect claims and other attributes, its configurable user consent interface and consent-trimming functionality, its concept of "privileged scopes" for client credentials grants, its ability to perform per-client and/or per-resource/scope authorization restrictions on the basis of user information such as LDAP or other directory group membership, its ability to have the authorization server automatically derive scope authorization or non-authorization, its API for configuring non-password authentication factors, and its optional email-based authentication factor.

As for Organization B, the main contributions toward Tomiko's design were its ability to replace or modify user interaction pages, interfaces, and components, its ability to complete configure or replace LDAP queries so that they are not reliant on any particular LDAP implementation or protocol extensions, its ability to offer user-driven profile attribute management and password (and password policy) management, its ability to allow the processing of non-textual profile attributes such as profile images, its generic extensions to support multiple authentication factors, its API and authentication mechanism to support the creation of external authentication factors, its reference SSH-based authentication factor, its abil-

ity to use WebSockets for server-driven authentication factor events (such as rolling TOTP codes or waiting for an SSH connection to be made), and its implementation of the OpenID Connect Dynamic Registration [3] specification to support mail clients wanting to make use of SASL mechanisms like OAUTHBEARER or XOAUTH2 for authentication.

Organization C's contributions include the simplification of Tomiko's deployment and hosting model (and the ability to make the previously hard dependency on an OCI-container runtime optional), its non-reliance on any particular shell or shell script to install or bootstrap, its ability to be configured by either configuration file or environment variables depending on the hosting method chosen, its JSON-file based user directory support, and a reference implementation of an external WebAuthn authentication factor.

Chapter 5

Comparisons and Related Work

This chapter compares Tomiko to existing software in the same space (authorization servers, identity providers, and similar software). For each comparison, a brief description of the software is given alongside some of the software's limitations and how Tomiko addresses those limitations is explained. Such comparisons are relevant because before developing Tomiko, a search was conducted for a piece of software that could flexibly meet the requirements of Organizations A and C and none were found to be satisfactory.

Later, the chapter describes why Tomiko uses OAuth 2.0 and OpenID Connect rather than other available protocols and additional usability features that influenced Tomiko's overall design.

5.1 Keycloak

Keycloak[16] is a popular free authorization server and identity provider currently maintained by Red Hat. The software is written in Java and currently contains about 85,000 lines of

code. Keycloak supports both OAuth 2.0 and OpenID Connect and SAML, is configurable both at runtime through a web interface and via configuration files. It can be extended by writing and loading Java extensions.

Keycloak is a very competent piece of software and could likely have met the organizations' needs. Unfortunately, a few aspects of Keycloak made is inconvenient to use. The most limiting aspect is that Keycloak is completely monolithic. The user interface, the user directories, the attribute mappings, the authentication factors and all configuration are within the Keycloak application. While it's possible to extend the Keycloak application with custom behaviours, these behaviours must be implemented in Java and loaded into Keycloak into predefined hooks.

Organization A would have required many extensions to support its workflow (custom attributes, multiple user data sources, special client privileges, protected scopes, etc). Implementing all of these in Keycloak would have required, for each extension, each contributor to have a significant understanding of OAuth 2.0 and OpenID Connect and its vulnerabilities and best practices. Some extensions (such as custom secondary authentication factors) would either been impossible to implement or required using such low-level APIs that it would have been more difficult to implement them *in* Keycloak than *outside* Keycloak.

Keycloak supported the WebAuthn standard used by Organization C, but did not support the format of their existing key enrollments or support exporting key enrollments to other systems in the future. Keycloak also did not support reading user profiles from JSON files without building this feature in an extension, and even with an extension disabled much of Keycloak's functionality in unpredictable ways.

5.2 Hydra

Hydra[17] is a popular free OAuth 2.0 and OpenID Connect authorization server/identity provider developed by Ory. Hydra's implementation of OpenID Connect is certified by the OpenID Foundation. Hydra supports a similar extensibility model to Tomiko, where an intermediary API is used to develop custom "endpoints" for login and consent.

Like Tomiko, Hydra partly processes authorization requests, then generates a challenge and redirects the user to an endpoint of their choosing with a challenge ID in as a query parameter. The custom application at that endpoint extracts the query parameter and gets challenge information from Hydra using an alternate channel. It authenticates the user in any way, then instructs Hydra to accept or reject the authorization request and continue the login flow. Hydra then redirects the user to a consent endpoint which works in a similar way.

This structure is similar to the way Tomiko's components are connected, but with a few key differences. When using custom login or consent endpoints, Hydra leaves almost every aspect of authorization entirely up to the implementation. This is extremely flexible, but means that between similar but non-identical implementations, everything must be entirely re-implemented or shared via custom libraries. There is no inherent imposed or even recommended structure to the implementation of these endpoints, which means developers again need a deep understanding of OAuth 2.0 and OpenID Connect, their vulnerabilities, and current best practices in order to implement them in a safe way. Since the APIs for integrating these pages are so simple, there's no straightforward way of sharing "building blocks" for these pages with the community either. Unlike Tomiko which has a standard API for authentication factors, Hydra only has a simple API to accept or reject an entire grant, so

factors must be implemented from scratch for each implementation. Tomiko components' APIs are much finer-grained and, while entire major components are designed so that they can be entirely replaced if necessary like Hydra endpoints, for more standard cases they also offer a safe basic structure with simpler extension points that can be used without knowledge of the entire system and ecosystem.

Another strange design decision in Hydra is the way of accessing its API sockets. `tomiko` binds a single socket to host both the "internal" and "external" parts of its API, but Hydra binds two sockets. One socket is meant for external use by clients and contains the usual OAuth 2.0 based access control for its endpoints/actions. The other socket is its administrative API, meant for internal use by custom endpoints accessing its API to address challenges. This socket does not have any access control and it meant to be protected at the network layer by a firewall or by an external API gateway adding a separate level of custom access control policies. Some portions of the internal and external API require the use of authentication in the form of custom static API keys, but this is a very strange choice for a piece of software whose entire purpose is issuing authorization tokens according to standard OAuth 2.0 mechanisms. This seemed to suggest that in the design of Hydra, its own token issuing and validation process were in some way considered insufficient for use between its own components. In contrast, Tomiko uses a single socket protected by its own standard OAuth 2.0 mechanism between all its components, usually in the form of standard client credentials grants with protected scopes.

5.3 Security Assertion Markup Language (SAML)

While Tomiko is built to be an identity provider to be used with OAuth 2.0 and/or OpenID Connect compliant software, these are not the only standard federated identity protocols in use today by organizations. The other main standard is Security Assertion Markup Language (SAML) 2.0, a protocol for authentication and authorization standardized by OASIS in 2005 [20].

In order to ease adoption and support as much software as possible, many identity providers support both OAuth2.0/OpenID Connect and SAML 2.0. Tomiko explicitly does not and will never implement SAML 2.0 for a few reasons.

Issues SAML 2.0 is a notoriously difficult standard to implement correctly and securely. Since its inception in 2005, there have been a plethora of vulnerabilities found with implementations of SAML 2.0, many of which were so bad that they completely broke the protocol. While the core SAML 2.0 standard is not all that complicated, it builds on some previous standards that are very problematic.

In SAML 2.0, messages between parties are signed and (optionally) encrypted by the sender according to a specific algorithm or standard. Because the message format and content as well as the format and content of the signatures and encryption methods must be carefully controlled, these typically transport-layer concerns must be handled at the application layer, which means there is a very large potential security vulnerability surface that every application participating in a SAML 2.0 exchange must carefully cover.

Messages are transmitted in XML, and signed according to the W3C's XML Signature specification from 2002, also known as XML-DSig [14]. Despite SAML 2.0's ubiquity, the

degree to which the correctness of XML-DSig is crucial to the utility of SAML 2.0, and XML-DSig existing for over 20 years, the W3C's interoperability report on XML-DSig reveals that among all known implementations of the standard, only *two* of them even have enough overlap in feature coverage to be even *theoretically* interoperable with each other [15]. The reason for this is that XML-DSig is extremely complicated to implement, extremely complicated to use correctly, and fundamentally flawed in design due to complected layers of the OSI model being mixed together.

As a result, even a completely correct identity provider implementation, with 100% test coverage, and 100% security audit coverage fails completely to provide secure authentication and authorization when *any* participant in a SAML 2.0 exchange has an incorrect or vulnerable implementation. This is of course not a desired property of the central piece of an organization's identity and security infrastructure. To partially mitigate this, most identity providers "supporting" SAML 2.0 today actually only support a loosely/informally defined subset of SAML 2.0, opting out of many of its core features for the sake of interoperability. Even so, most software does not work particularly well together unless both sides of an exchange are using the de-facto "standard" implementation of XML-DSig, `libxmlsec1` [18]. Since this is a C library, this also requires that these applications or their dependencies eventually interoperate with C somehow, which opens up a whole other avenue for security vulnerabilities if memory is managed unsafely somewhere in the process. Using 'libxmlsec1' also requires using a specific XML serialization and deserialization library, because XML-DSig relies on "canonically" representing XML documents.

Obsoleteness Related to the previous point, if identity providers are extending their protocol offerings to SAML 2.0 in order to be more interoperable with software relying on identity assertion, and the safest bet at an interoperable implementation of SAML 2.0 is to use a stripped down subset of it and hope the software you exchange messages with makes the same implementation choices you did, then perhaps SAML 2.0 (as implemented by most software) does not meet its design goals by actually offering any particular reason to actually use most of its features.

Being more recently standardized protocols, OAuth 2.0 and OpenID Connect have had the opportunity to learn from many of the weaknesses of SAML 2.0's design. The OAuth 2.0 standard wisely relies on the underlying transport and network layers to provide encryption of messages in transit and authentication of participants and instructs implementors to treat tokens as opaque strings to avoid being dependent on a particular format or encoding.

The OpenID Connect protocol does include encryption for some components in the form of JSON Web Tokens[21], but the JSON Web Token standard is far simpler and more widely implemented than XML-DSig because it does not require canonicalization and does not store signatures inline with signed data and therefore does not require any particular parsers.

Together, both these standards offer all the promised functionality of SAML 2.0 in a way that can actually be relied upon to be interoperable, and the OpenID Connect Discovery and JSON Web Key specifications allow software that do not make the same implementation choices and cannot work together to detect these circumstances and decide whether to refuse to work together or negotiate a useful common subset.

5.4 Usability, Security, and Compliance

5.4.1 Hosting

Tomiko was designed to be simple to host, so it was an important design constraint that the process of getting a Tomiko instance up and running (with all the reference components) was a frictionless process. Commercial identity providers either cannot be self-hosted (they are available as SaaS offerings only) or have long and complex setup processes. For organizations that expect many users or requests to the identity provider, those offerings also become difficult to scale. SaaS solutions might slow down or require more expensive service plans to accommodate additional request volume, and self-hostable offerings might only be vertically scalable by upgrading the hardware or virtual resource allocations.

In contrast, Tomiko can be deployed with a single command using docker-compose or Kubernetes manifests distributed with the software. If the default deployment layout (consisting of a single instance of each of the reference components) does not meet your needs, the manifests can be modified/configured to add additional replicas or an entirely custom deployment can easily be planned since each component can also be deployed in isolation with a single command. Each component has default configurations where possible, and does not require a setup wizard or configuration step – they are ready to serve requests as soon as they are up and running.

The ability to scale out horizontally by adding replicas required Tomiko's reference components to be designed in a way that allows sequential requests to components to go to different instances or have instances be removed or restarted without impacting the overall state of the system. To achieve this, Tomiko components do not store transaction or coordi-

nation information in memory and communicate through HTTPS (which can be transparently load-balanced and has well-defined caching and request-retrying semantics) or MQTT message queues which have well-defined delivery semantics.

If caches or message queues are used, the management of consistent system state via coordinated message-passing and the management of coordinated caches are both handled by robust existing components outside the Tomiko system (a reference deployment with these features uses Redis and RabbitMQ) so that organizations that have already operationalized such software and understand their behaviours can reuse their instances, tooling, or institutional knowledge.

As for persistent state, instances of Tomiko reference components do not store data that is expected to be referenced between multiple different requests in-memory. Such state is persisted using the configured persistence connectors (in most cases, to a relational SQL database). This allows the underlying persistence layer (which, again, organizations might have already operationalized) to ensure that all component instances see a consistent view of the system's state at all times. Modifications to this state are designed to be free from data races that might affect correctness (such as allowing an access code to be reused on two different component instances, or allowing an expired refresh token to be used while or before a different component instance invalidates expired tokens in the database).

These design decisions mean that it is always safe to add or remove instances of any reference component instance without reconfiguring the system or causing downtime or service interruptions. Even for smaller organizations with simple deployments where this is not needed, ensuring that these properties are present within the system still allows for straightforward hosting of Tomiko components using cloud-native technologies (such as con-

tainerization, Kubernetes, and for some components even serverless platforms) which make Tomiko easier to host.

For each component where it makes sense to, the reference implementations expose health and readiness endpoints as part of their APIs, which monitoring and orchestration software can use to manage deployments and instance lifecycles, and backend components use instrumentation and logging according to OpenTelemetry specifications, which allow for metric collection and behavioural observability in most third-party monitoring tools.

Tomiko is also designed to not require any particular network topology and does not make any assumptions about reachability, segmentation, network policies, or firewalling to maintain its correctness guarantees. This is in contrast with other identity providers that expose unprotected management APIs that other network layers must restrict the use of in order to secure the system.

5.4.2 Argon2 Hashing

As security software, Tomiko must take great care to not introduce unnecessary vulnerabilities into the authentication or authorization processes. Tomiko's reference components' stateful features are carefully designed so that even if an attacker gained access to a copy of *tomiko*'s database, they could not undermine Tomiko's overall security guarantees.

A large part of this protection is based on the fact that no secrets are stored in *tomiko*'s database. All secrets (including transient/temporary ones) are verified through salted hash matching. This means a copy of the database could not cause client secrets, refresh tokens, token revocation secrets, or even pending authorization codes to be discovered. The private

keys used to encrypt and/or sign tokens and the secrets for decrypting those private keys are not stored in a database and the decrypted key is only ever stored in memory for the lifetime of the process.

Tomiko's reference components use the Argon2 hashing algorithm[19]. This algorithm won the 2015 Password Hashing Competition and is designed to be resilient to GPU cracking attacks and various side-channel attacks. Since the Argon2 hashing algorithm has configurable parameters, *tomiko* configuration options are available to choose the values for these parameters (such as the number of memory passes). If left unconfigured, *tomiko* uses safe defaults for most parameters and performs a short calibration to determine a suitable value for the number of hash iterations. Since most interactions with Tomiko involve authentication (with client authentication, token authentication, or the exchange of authentication codes for tokens), secret hashing is also used as a form of rate limiting. *tomiko*'s startup calibration routine tries to find a value for the number of hash iterations that makes hash checking for values of approximately the same length as the secrets the system is expected to receive and check take between 250 and 1000 milliseconds on the system it is running on.

5.4.3 Compliance

The reference Tomiko components are compliant with (at least) the following (non-exhaustive) list of specifications

- IETF RFC 6794: The OAuth 2.0 Authorization Framework [1]
- IETF RFC 8252: OAuth 2.0 for Native Apps [22]
- IETF RFC 7636: Proof Key for Code Exchange by OAuth Public Clients [23]

- IETF RTC 7519: JSON Web Token (JWT) [21]
- IETF RFC 8707: Resource Indicators for OAuth 2.0 [24]
- IETF RFC 9207: OAuth 2.0 Authorization Server Issuer Identification [25]
- IETF RFC 9068: JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens [26]
- IETF draft-ietf-oauth-v2-1-08: The OAuth 2.1 Authorization Framework *(partial)* [27]
- IETF draft-ietf-oauth-rar-23: OAuth 2.0 Rich Authorization Requests *(partial)* [28]
- IETF draft-ietf-oauth-security-topics-22: OAuth 2.0 Security Best Current Practice [29]
- OpenID Connect: Core [2]
- OpenID Connect: Discovery [4]
- OpenID Connect: Dynamic Registration [3]
- OpenID Connect: OAuth 2.0 Multiple Response Types [5]
- OpenID Connect: RP-Initiated Logout [6]
- OpenID Connect: Session Management *(with some non-standard modifications)* [7]
- OpenID Connect: Front-Channel Logout [8]
- OpenID Connect: Back-Channel Logout [9]

Chapter 6

Conclusion

In conclusion, Tomiko is a safe, modern, and extensible OAuth 2.0 and OpenID Connect authorization server/identity provider that has been demonstrated to interoperate with a large number of pieces of third-party software that use these protocols for delegated or federated authentication and/or authorization.

Tomiko implements various IETF and OpenID Foundation standards, best practices from currently accepted and draft recommendation documents and specifications, and is even already compliant with the current draft of the OAuth 2.1 specification.

Though its production use by different organizations, it has shown that it is usable and useful in a variety of different situations and to different populations with different use-cases.

It is significantly different from existing authorization servers/identity providers, addresses many of their limitations, and is freely licensed under the GNU General Public License v3.0 so that its contributions may be freely used directly or as inspiration for improvements in existing or new systems in the future.

Bibliography

[1] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749.

[2] N. Sakimura. *OpenID Connect Core 1.0*. URL: https://openid.net/specs/openid-connect-core-1_0.html.

[3] N. Sakimura. *OpenID Connect Dynamic Client Registration 1.0*. URL: https://openid.net/specs/openid-connect-registration-1_0.html.

[4] N. Sakimura. *OpenID Connect Discovery 1.0*. URL: https://openid.net/specs/openid-connect-discovery-1_0.html.

[5] B. de Medeiros. *OAuth 2.0 Multiple Response Type Encoding Practices*. URL: https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html.

[6] M. Jones. *OpenID Connect RP-Initiated Logout*. URL: https://openid.net/specs/openid-connect-rpinitiated-1_0.html.

[7] B. de Medeiros. *OpenID Connect Session Management*. URL: https://openid.net/specs/openid-connect-session-1_0.html.

[8] M. Jones. *OpenID Connect Front-Channel Logout*. URL: https://openid.net/specs/openid-connect-frontchannel-1_0.html.

[9] M. Jones. *OpenID Connect Back-Channel Logout*. URL: https://openid.net/specs/openid-connect-backchannel-1_0.html.

[10] J. Pizzuco. *SSHAFT*. URL: https://gitub.com/toshokan/sshaft.

[11] Mozilla Thunderbird Contributors. *Thunderbird:Autoconfiguration:ConfigFileFormat*. URL: https://wiki.mozilla.org/Thunderbird:Autoconfiguration:ConfigFileFormat#OAuth2.

[12] Jim Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. June 2006. DOI: 10.17487/RFC4511. URL: https://www.rfc-editor.org/info/rfc4511.

[13] W3C. *Web Authentication: An API for accessing Public Key Credentials*. webauthn. URL: https://www.w3.org/TR/webauthn/.

[14] W3C. *XML Signature Syntax and Processing Version 1.1*. xmldsig-core-1. URL: https://www.w3.org/TR/xmldsig-core1/.

[15] W3C. *XML-Signature Interoperability*. URL: https://www.w3.org/Signature/2001/04/05-xmldsig-interop.html.

[16] Red Hat. *Keycloak*. URL: https://www.keycloak.org/.

[17] Ory. *Hydra*. URL: https://www.ory.sh/hydra/.

[18] Aleksey Sanin. *XML Security Library*. URL: https://www.aleksey.com/xmlsec/index.html.

[19] Jean-Philippe Aumasson. *Password Hashing Competition and our recommendation for hashing passwords: Argon2*. URL: https://www.password-hashing.net/.

[20] Organization for the Advancement of Structured Information Standards. *Security Assertion Markup Language (SAML) v2.0*. 2005.

[21] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[22] William Denniss and John Bradley. *OAuth 2.0 for Native Apps*. RFC 8252. Oct. 2017. DOI: 10.17487/RFC8252. URL: https://www.rfc-editor.org/info/rfc8252.

[23] Nat Sakimura, John Bradley, and Naveen Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. Sept. 2015. DOI: 10.17487/RFC7636. URL: https://www.rfc-editor.org/info/rfc7636.

[24] Brian Campbell, John Bradley, and Hannes Tschofenig. *Resource Indicators for OAuth 2.0*. RFC 8707. Feb. 2020. DOI: 10.17487/RFC8707. URL: https://www.rfc-editor.org/info/rfc8707.

[25] Karsten Meyer zu Selhausen and Daniel Fett. *OAuth 2.0 Authorization Server Issuer Identification*. RFC 9207. Mar. 2022. DOI: 10.17487/RFC9207. URL: https://www.rfc-editor.org/info/rfc9207.

[26] Vittorio Bertocci. *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. RFC 9068. Oct. 2021. DOI: 10.17487/RFC9068. URL: https://www.rfc-editor.org/info/rfc9068.

[27] Dick Hardt, Aaron Parecki, and Torsten Lodderstedt. *The OAuth 2.1 Authorization Framework*. Internet-Draft draft-ietf-oauth-v2-1-08. Work in Progress. Internet Engineering Task Force, Mar. 2023. 88 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/08/.

[28] Torsten Lodderstedt, Justin Richer, and Brian Campbell. *OAuth 2.0 Rich Authorization Requests*. Internet-Draft draft-ietf-oauth-rar-23. Work in Progress. Internet Engineering Task Force, Jan. 2023. 45 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-rar/23/.

[29] Torsten Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-22. Work in Progress. Internet Engineering Task Force, Mar. 2023. 60 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/22/.