Designing a Secure Client for MIT OpenID Connect

Huy Dai
huydai@mit.edu

Bella Xu
bellaxu@mit.edu

Joey Zheng
joeyz@mit.edu

Github Repo:
https://github.com/mit-oidc-client/mit-oidc-client

Abstract

There are a growing number of web services built at MIT each year, many of which are made by student developers for the MIT community. Often, it is necessary for those services to offer provide a secure and easy-to-use authentication option for their users. One option is to use MIT OpenID Connect Pilot (OIDC) service, which integrates with MIT Touchstone for authentication. However, we found there to be a knowledge gap and technical barrier that prevents its widespread adoption. More than this, the OpenID protocol is known to have some security flaws. In our project, we provide an easy-to-use template along with a comprehensive documentation for MIT students looking to develop secure web services that integrate with the MIT OpenID Connect service, as well as supplement the security through implementing the OpenPubKey extension for message signing and verification.

Contents

1 Background

1.1 What is OpenID Connect?

OpenID Connect is a protocol built on top of OAuth2.[8] It is important to note the difference between the two. OAuth2 provides authorization, or permission, to access certain resources from one application to another on behalf of the Resource Owner (RO) - usually the users of the application. On the other hand, OIDC provides authentication, or identity of the Resource Owner.

In this paper, we are concerned with the authentication flow for OIDC that follows the "PKCE (Proof Key for Code Exchange) Authorization Code Flow". To begin, the client application must generate:

1. a random $\texttt{nonce} \leftarrow^r \{0,1\}^\lambda$,
2. a random PKCE Code Verifier $\texttt{CV} \leftarrow^r \{0,1\}^\lambda$,
3. and a PKCE Code Challenge, $\texttt{CC} \leftarrow^r \texttt{SHA256}(\texttt{CV})$

for a security parameter λ. The `nonce` prevents replay attacks and the `CV` and `CC` creates a one-time challenge.

With this set up, the client sends the `CC`, `nonce`, Client ID, and redirect URI to the authorization server. On the authorization server, the Resource Owner will authenticate themselves and set the scope, or permissions, for the client. Then, returned to the client is the authorization code. The client then sends the `CV`, authorization code, Client ID, and client secret to an endpoint on the resource server. The resource server then verfies that $\texttt{CC} = \texttt{SHA256}(\texttt{CV})$. If so, the authorization code is exchanges for an access token and ID Token (and optionally a refresh token). The access token is used to access resources from the resource server. Crucially, the ID Token is a JWT that can be decoded to extract authenticated information such as the name and email of the Resource Owner.

1.2 What is OpenPubKey?

OpenPubKey is an augmentation to OpenID Connect published in early 2023. In this section, we describe OpenPubKey's purpose and highlight main points of its protocol.

Prior to OpenPubKey, OIDC functioned under Bearer Authentication, and there was no secure method for users to sign messages with their OpenID identity. Created to remedy these issues, OpenPubKey cryptographically binds a user's public key to their ID Token from an OpenID provider by creating a PK Token. The private key can now be used for signing messages, and with the PK Token, the messages can be attributed to the OpenID Connect identity. Through doing so, authentication is upgraded to Proof-of-Possession, which protects against various security flaws and reduces trust assumptions [4].

1.2.1 Generating PK Tokens

The OpenPubKey protocol makes a few minor adjustments to the OIDC protocol. We first detail the steps involved in creating the PK Token.

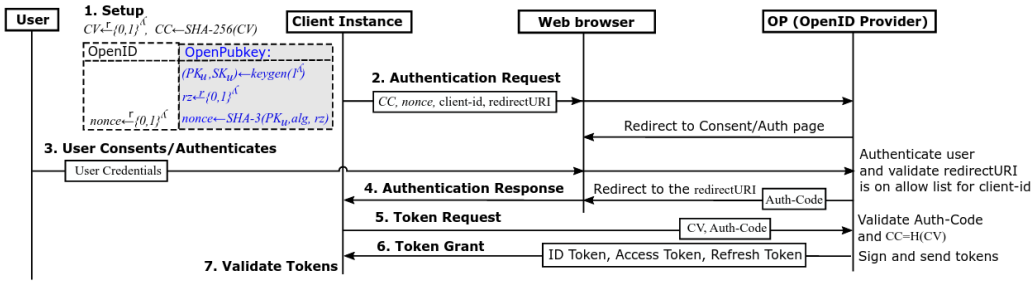

Figure 1: Standard OIDC Authorization Code Flow, with changes for OpenPubKey in blue. Source: [4]

PK Token creation takes advantage of the requirement for the client instance to send a nonce, and for the OpenID provider (OP) to return the signed nonce as part of the ID Token. Using this, we can inject the user's public key as part of the nonce, thereby having the ID Token bind the user's public key to their OIDC identity.

The user first generates a pair of ephemeral keys to be used for the session. Then, instead of sending in a random nonce as in the standard OIDC protocol, we send the hash of the public key and a random value as the nonce. More specifically, the user creates a Client Instance Claim

$$cic \longleftarrow (pk, alg, rz)$$

where pk is the public key, alg is the algorithm name, and rz is a randomly generated value, and submit the SHA-3 hash of the cic as the nonce

$$nonce \longleftarrow \texttt{SHA-3}(cic).$$

Then, following the steps of the OIDC protocol, the OpenID provider returns the ID Token with the nonce the user passed in, meaning that the returned ID Token contains the OP's signature on the nonce which commits to the user's public key. To complete creation of the PK Token, the user signs the returned ID Token and the cic as the protected header with their signing key.

Since only client side changes are made, OpenPubKey functions seamlessly with existing OpenID Providers with no change on their part. The nonce change faces no issues since the OP is bound to return the nonce it received without change, and the inclusion of the random value in the nonce allows it to still function as a random nonce. While the returned ID Token with the custom nonce binds the user's public key to their OIDC identity, the additional user signature prevents attackers from binding their identity to someone else's public key (Identity Misbinding Attacks or Unknown Key-Share attacks). [4]

```
pkT ← {
  "payload":idT.claims ←{
    "nonce":SHA-3(cic ← (PK_u,alg,rz)),
    "iss":issuer,
    "exp":expires on,
    "iat":issued at,
    "aud":audience,
    "sub":subject ID,
    "email":subject email
  },
  "signatures":
  [{
    "signature":σ_o ← SIGN(SK_o,(idT.claims,h_o)),
    "protected":h_o ←{"alg":"RS256",
      "typ": "JWT", "kid":"123"},
  },
  {
    "signature":σ_u ← SIGN(SK_u,(idT.claims,h_u)),
    "protected":h_u ←{
      "upk":PK_u,
      "alg":alg,
      "rz":rz
    },
  },
```

Figure 2: PK Token structure, where the payload is ID Token payload (`idT.claims`) and for the signature, secret key, and header, the subscript o denotes originating from the OP, and the subscript u denotes from the user. Source: [4]

1.2.2 Verifying PK Tokens

PK Token verification follows a simple procedure reversing the steps above. We first check that the Client ID, audience (aud), and issuer (iss) are as expected. Then, we verify the ID Token is valid under the OP public key. Next, we check that the nonce claim is equal to the SHA-3 hash of the cic included in the protected header. Then, we check that the cic is valid and that the public key contained in it verifies the user signature over the PK Token. If any of these steps fail, the PK Token is rejected. [4]

1.2.3 Generating OSMs

With PK Tokens, we can now generate and verify messages—specifically OpenPubKey Signed Messages (OSM), which are signed with the user's public key and can be attributed to their OIDC identity through their PK Token. OSMs are structured as a variant of JSON Web Signatures (JWS) with a few required header claims, being which the $typ =$ "osm", the $alg =$ same alg as the one in the cic. Finally, the kid must also be the SHA-3 hash of the user's PK Token. These features together bind the OSM to the user's identity and ensures the security of the OSM against algorithm switching attacks. [4]

1.2.4 Verifying OSMs

On receipt of an OSM and its corresponding PK Token, we verify as follows. We first assert that the *typ* claim is "osm", the *alg* claim matches the *alg* in the cic of the PK Token, and that the *kid* commits to the PK Token via SHA-3 hashing. Then, we verify that the PK Token is correct via the PK Token verification procedure above. Finally, we check that the signature on the OSM verifies under the user's public key in the PK Token. [4]

1.2.5 OpenPubKey PoP Authentication

One important feature of OpenPubKey is the ability to do proof-of-possesion authentication using the challenge-response framework. To do this, the verifier sends a random value to the client. Then, the client includes the random value as part of an OSM and returns it to the verifier. The verifier can then verify the OSM is valid and contains the random value, which shows proof-of-possession of the signing key. This is an upgrade from standard OIDC Bearer Authentication, where the client must show the verifier the ID Token. Proof-of-possession helps prevent against token replay attacks, as well as various other vulnerabilities OpenID Connect faces. [4]

OpenPubKey also consists of various other important features, such as cosigners and multifactor authentication, but we consider them to be beyond the scope of this project.

2 Related Work

2.1 Authentication Systems at MIT

At MIT, there are multiple authentication systems and methods that are frequently used. Each of them offers their own strengths and drawbacks, which we will summarize below:

1. **Touchstone** is MIT's proprietary implementation of the Shibboleth system, which offers a single sign on system (SSO) for web applications. It is the most flexible, allowing users to either log with MIT certificate, kerberos, or a Collaboration account (for non-MIT collaborators). However, Touchstone requires special software - namely the Shibboleth SP packages - and custom configuration changes to run, and are largely seen in MIT-controlled websites and the IS&T wiki.

2. **Shimmer** is CSAIL's implementation of the Touchstone system integrated with OpenID Connect (OIDC). While it utilizes OpenID Connect and is the authentication system of choice for many CSAIl services and Course 6 websites, its usage is limited to CSAIL account holders, which makes it unavailable for MIT students to use more widely.

3. **MIT OpenID Connect Pilot (OIDC)** is run by MITRE as part of a collaboration effort with MIT KIT starting in 2014, and is also based on the OpenID Connnect

protocol.[9] This service also offers a dynamic registration endpoint like Shimmer, except it is open to use by anyone at MIT with a Touchstone account. Its main issues are that the servers running the system seem to somewhat unreliable, causing unexpected timeout issues at different times of the time. Students have also complained about its difficult-to-read documentation [6] [8] and lack of example client as reasons hindering its adoption.

4. **Certificates**, namely MIT X.509 client certificates, are frequently used as a standalone authentication option for some MIT web services, including ones hosted on Scripts, which is popular MIT hosting service made by SIPB for students to use. The main issue with having certificates as the standalone certification option is that it is difficult to install certificates on mobile devices, and are not very extensible to MIT student web services running on third-party hosting solutions like Heroku, Render, or standalone VMs.

Out of the auth systems we surveyed, MIT OIDC [9] seemed to be the most promising for integrating into MIT student web services. Not only is MIT OIDC the recommended authentication system by IS&T,[5] its current issues seem to be with lack of student awareness about its existence and lack of an easy-to-use client implementation rather than limitations by the system itself.

2.2 Existing OIDC Libraries

Since OpenID Connect is a well-known protocol, open-source clients for it have been written, including:

1. **oidc-client-ts** is a Typescript library that provides OIDC and OAuth2 support for browser-based client applications [1]. The main issue we found with this template is that since all code is executed on the client side, it requires storing the *client_secret* in the frontend code. Because browsers are *public clients*, there is no way to guarantee the security of the secret. To fight against this, `oidc-client-ts` implements the **PKCE extension**, which came as part of the OAuth 2.0 protocol, and replaces the *client_secret* with a random string generated by the application for every auth code submission. Unfortunately, the MIT OIDC service does not currently support this extension, and without it, auth systems using this library would be vulnerable to impersonation attacks and potential leakage of user's data.

2. **passport-mitopenid** is a project written by a MIT student in 2018 that implements a sample Node.js application that works with MIT OIDC service using Passport.js [7]. This project is very similar to `node-openid-client`, which is the official package (certified by OpenID) for implementing OpenID Client and Relaying Party in Node.js applications. However, for both libraries, it is an incomplete solution, since they are intended for web applications running solely on Node.js, whereas for our solution we

want to have an integrated authentication workflow between a Node.js backend and a frontend running a web framework like React.js.

2.3 Other OpenPubKey Implementations (BastionZero)

OpenPubKey is created by BastionZero and is an important feature of their product. BastionZero uses the concepts in OpenPubKey to create a BastionZero OpenPubKey Token, signed by both the SSO and BastionZero. This token is then used to create an authenticated channel for users to work in as their company's service [2].

Despite being open source, BastionZero's implementation is understandably intricately linked with the rest of their service's code implementation. As such, the OpenPubKey implementation is only used through the BastionZero service and is not usable for other purposes. Other companies (like Google or Microsoft) use similar ideas in their SSO services, but their implementations are not open source or available for use with other OpenID providers. Our project aims to be the first open implementation of OpenPubKey, where it is implemented as a separable component of our client for MIT OIDC.

3 Contributions

3.1 Survey

We conducted a survey to gain insights into the needs of the MIT community and identify design decisions that would be most beneficial. A total of 61 responses were collected, revealing that some popular MIT web services, such as courseroad.mit.edu, use MIT credentials to log in, while many others, such as HackMIT, hydrant.mit.edu, and dorm databases, do not. The downside of using alternative methods such as password-based authentication, as in the case for HackMIT, is the complexity of a secure password management system. Moreover, users may encounter instances of not receiving verification emails and needing to contact admins for registration and password resets.

To address these issues, the MIT OIDC [9] service can streamline this process of authentication and retrieval of user information using MIT credentials. Of course, this is only available to MIT community members, but MIT students make up a good portion of hackers each year. For services specifically targeting and used by MIT community members, such as hydrant.mit.edu and dorm databases, using MIT credentials instead of SSO with Google would be more suitable.

Our survey also showed that MIT community members are the most comfortable sharing their kerberos to be used in web applications, and that student developers frequently use Typescript as their web programming language of choice. These insights are reflected in our implementation decisions. Additionally, the survey revealed a need for better documentation on setting up and integrating the OIDC workflow. This prompted us to thoroughly document our implementation and design choices in order to make MIT OIDC more accessible to MIT

web service developers.

3.2 OIDC Client

3.2.1 OIDC System Diagram

The follow diagram summarizes our system interactions between the frontend, backend, and the OIDC server:

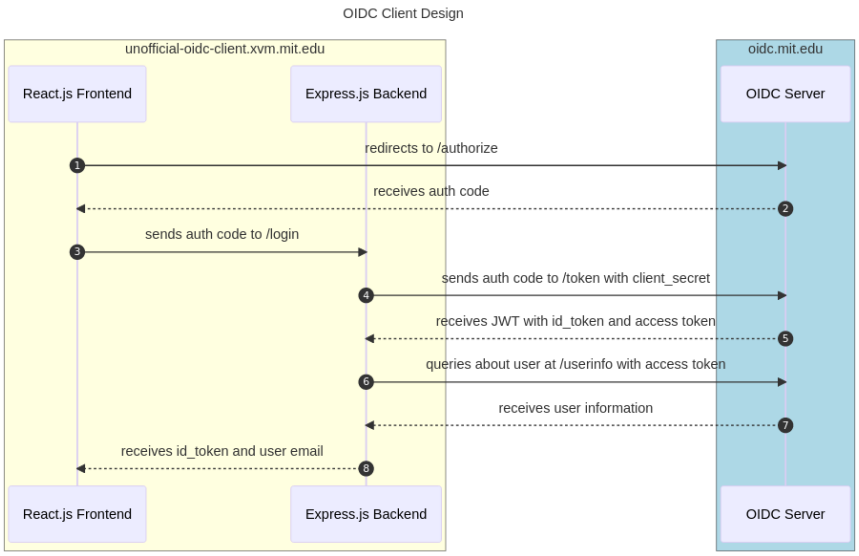

Figure 3: Authentication Workflow

We can see the design of the system closely follows the authentication flow for the OpenID Connect protocol [8]. First, the user is redirected to the authentication endpoint, where they log in. The frontend then receives an auth `code`, which it validates and passes to the backend. The backend server then is responsible for exchanging this information for an ID Token and access token. It uses the access token to get information about the user, namely their email, and returns the `(id_token, email)` pair back to the frontend.

3.2.2 Frontend

The primary structure of our frontend code is as follows:

```
1 <AuthProvider>
2   ...
```

```
  <Routes>
  <Route element={<Layout />}>
    ...
    <Route path="/login" element={<LoginPage />} />
    <Route path="/oidc-response" element={<OidcResponseHandler />} />
    <Route
      path="/protected"
      element={
        <RequireAuth>
          <ProtectedPage />
        </RequireAuth>
      }
    />
  </Route>
  </Routes>
</AuthProvider>
```

In App.tsx, we define an auth context manager `<AuthProvider>`, which keeps track of the current logged in user. It provides a property *auth.user* through `useAuth()`, which stores the email of the current user as a string. We set the default value of this property to be an empty string.

To prevent displaying of certain component(s) until the user is logged in, we wrap them with the `<RequireAuth>` component. In this case, we want to hide the `<ProtectPage>`, which contains the chatroom. As noted in the security discussion section about authenticated actions, this only prevents browser rendering, but any actions done using these components need to be further authenticated in the backend (preferably using some form of user-identifiable session cookies).

In our frontend code, we provide two React router paths, namely `/login` and `/oidc-response`, to handle the OIDC logic workflow.

1. `/login` handles the redirect from the current website to the OIDC's authentication endpoint. It supplies the *client_id* and *redirect_uri* to the OIDC server - in addition to other required and optional fields - as query parameters.

2. Once the user successfully authenticates to the OIDC server, they are redirected to the `/oidc-response` endpoint on the webpage. The browser takes in the authorization code in the URL, verifies the returned state parameter is correct, and forwards the auth code to the backend using the backend's POST `/login` API endpoint.

3. The backend returns a response of type `loginResponse`, which looks like the following:

```
interface loginResponse {
success: boolean; //Whether or not the login succeeded
error_msg: string; //If failed, provide error message.
                   //Else, empty string.

//If success, these values should be populated.
id_token: string;
email: string;
}
```

The frontend then checks the `success` parameter. If it is true, then it stores the *id_token* to localStorage and signs the user in with `email` using the `auth.signin()` function. Otherwise, it outputs the error message to the user via the `<OidcResponseHandler/>` React component.

4. The webpage then reloads to show a "Welcome user_email@mit.edu!" message at the top (via `<AuthStatus/>` component), along with a signout button. The user can then access the Protected page, which in this case allows them to interact with the OpenPubKey-enabled chatroom.

3.2.3 Backend

The Express.js backend requires only one API endpoint for OIDC, which we define as a POST request.

```
app.post("/api/login", handleLogin);
```

The server takes the *code* variable and sends it to the `/token` endpoint on the OIDC server to exchange for an ID Token. It also includes the *client_id* and *client_secret* as HTTP Authorization Headers, along with a fixed *grant_type* and *redirect_uri*. **Note:** This step is the reason why the ID Token exchange is done on the backend, rather in the frontend, as the backend is considered a *confidential client* while the frontend is considered to be a *public client*. See the Related Works section on why we view this division in authentication process to be necessary.

Following that step, the backend server receives back an JSON object containing the ID Token from the OIDC server, which it then validates thoroughly. The checks are as follows:

1. Verify that the JSON object contains the necessary *id_token* and *scope* parameters. As part of the OpenID Connect protocol, it is possible for a user to successfully authenticate to the OpenID provider, but chooses to deny some or *all* of the requested scopes

asked by the application. For our model, since the ID Token and access to user's email is the bare minimum we need for our authentication process to work, we return with error message if *openid* or *email* scope is not provided.

2. Verify the *token_type* is of value "Bearer"

3. The ID Token is then taken from *id_token* field in the JSON object and has its signature validated. This is done by fetching the public key of the OIDC server at `/jwk` endpoint, convert it to PEM format, and using `jwt.verify()` from the jsonwebtoken library to verify the JWT's signature. The values from the ID Token are then exposed to us as a return value of this call.

4. We then check the *iss* (issuer) of the ID Token is "https://oidc.mit.edu", and that *exp* (expiration time) and *iat* (issued time) are correct relative to the current time.

5. We then verify our *client_id* is included as part of the *aud* list, and that the *nonce* in the ID Token matches the 'nonce' cookie sent by the browser.

Once we can be sure the ID Token is valid, we proceed with querying the `/userinfo` endpoint on the OIDC server for user information. Since we're solely interested in determining who the user that just logged in is, we decided to query for the *email* field. This action is done by the `getUserInfo()` function, which simply returns the user's email (or an error message if the lookup fails).

The backend then bundles the ID Token together with the user email, and sends it back to the user's browser.

3.2.4 Security Design

To ensure the security of our overall client framework, we made the following design choices:

1. HTTPS is required in development for both the front-end and back-end

    - HTTPS/SSL encryption ensures all communication between the user and the web service is protected. This is a hard requirement for safe-handling of OAuth ID and access tokens

2. Dependency on secure cryptographic libraries

    - When generating randomness or relying on cryptographic primitives like hash functions, we use secure libraries like built in browser's `Crypto.getRandomValues()` and `Crypto.subtle.digest()` functions, along with official JWT libraries. This ensures that we are generating secure randomness in our application and depends only on secure primitive implementation for signing and verification of data.

3. Utilization of state and nonce in authorization request

- As part of the OpenID Connect Basic Implementer's Guide 1.0, it is optional, but recommended, to implement the state and nonce parameter in the authentication request flow. The state parameter is used to mitigate against Cross-Site Request Forgery (CSRF, XSRF) attacks, and the nonce parameter help mitigate against replay attacks.
- For our application, we decided to implement both of these variables, storing them in as a variable in localStorage and a cookie, respectively. Both contain high levels of entropy (2^{128}) for security. Note: In the optional OpenPubKey flow, the nonce is replaced by a different format than a random string.

4. Secure handling of cookies

    - Whenever cookies are used in our implementation, namely the state parameter, we secure it by setting security parameters including:
        i `path`: Restricts sending the cookie to a specific API endpoint on the backend only
        ii `sameSite`: Disallows sending cookie on cross-site requests
        iii `secure`: Forces cookie to be sent over secure connections (HTTPS)

5. Safe type checking using Typescript

    - At MIT, especially in course 6.031, we promote the user of Typescript over Javascript because it allows for static typing, which helps code be more readable, bug-safe, and more easily maintainable. It's also the language of choice for most MIT students creating web services.

3.2.5 Discussion of Login System + Authenticated Actions

An important thing to note is that in our example React app, we require the user to authenticate to be able to access the content of the "Protected" page. However, this measure simply *hides* the chatroom React component, but doesn't actually prevent a malicious user from sending messages to the backend.

For example, a bad actor can call the `auth.signin()` function themselves in the browser's console, provided with a fake email to simulate a user login and see the chatroom. Alternatively, they can directly send a POST request to the backend `/api/messages` endpoint with a message of their choice.

Thus, it's important when developing an app that has a login system such as this one that you have a method of authenticating user actions *after* they have logged in. Indeed, in our Future Works section, we discuss the use of adding a session management system that allows for the backend application to keep track of logged in users via **session cookies** and verify that a user provide a valid session cookie whenever they want to perform an authenticated action.

3.3 OpenPubKey Extension

As part of the OIDC client implementation, we offer the optional feature of OpenPubKey. Included features are the ability to generate PK Tokens, verify PK Tokens, generate OSMs, and verify OSMs, where the implementation follows the steps explained in Section 1.2. If users choose to not include our OpenPubKey implementation, they can simply ignore it or disable it, since the changes to the OpenID flow are minimal and do not affect its function. All OpenPubKey code is implemented to the client side of our OIDC client. We decided to include OpenPubKey as part of our project to increase the security features of our client as well as to create a reusable implementation of OpenPubKey for others to use.

3.3.1 Implementation and Usage

To improve modularity of the system, we chose to place all OpenPubKey functions in its own file, and call them when necessary. Included in the file are type definitions for specific attributes like headers, parameter definitions for our `Crypto.subtle` functions, and helper functions for import/exporting keys between the `CryptoKey` objects used by our libraries and the JWK format that we save locally as well as hashing function helpers. We used some aspects of BastionZero's codebase to help develop our implementation, cited here [3].

To better understanding the workings of the code, we go through an example flow for Alice, an MIT student who wants to send messages in the chatroom to another student, Bob.

1. **Nonce Generation and Key Generation** As part of the auth flow for OIDC connect, Alice's client must send a nonce as part of its request to the OP. Instead of sending a random nonce value, we use `generateNonce()` to create the nonce for Alice containing her public key (as detailed in background section). If Alice does not currently have keys for her session, we generate keys with `crypto.subtle.generateKey()`, which returns a secret signing key and public verifying key. Both keys are exported to the JWK format, which is then converted to a base64 string and saved in localStorage. Now, we can generate the nonce by retrieving the public key in its current format as a base64 string, and placing it into the *upk* field of our *cic*, which is a JSON object that also contains the field *alg* ("ECDSA" in our case), and a random value rz that we generate. We save the cic as a JSON string to localStorage, and return the SHA-3 hash of the cic as the nonce.

2. **PK Token Creation** After the auth flow process returns the ID Token from the OIDC protocol, we follow the steps necessary to turn this into a PK Token in `generatePKToken()`. Reading from localStorage, we retrieve the private key as a `CryptoKey` object and our *cic* as a JSON object. We also extract the payload from the returned ID Token. Then, we use the private key to sign the payload of the ID Token along with the cic as the `userheader`. Finally, we construct our PK Token as a custom construction of `userHeader + "." + opHeader + "." + payload + "." + opSig + "." + userSig`, where 'user' refers to Alice's signature or cic, and 'op' refers to the OpenID provider's

header and signature. The newly generated PK Token is saved as-is in localStorage for convenience in generating OSMs.

3. **OSM Generation** At this point in the process, Alice is logged in to our site and has automatically been created a PK Token. Now, Alice can access the chatroom in the "Protected" part of our site. When Alice sends a message, we use `generateOSM()` to automatically attach Alice's signature & identity to the message. In this function, we get Alice's private key (as `CryptoKey`), her *cic* (as JSON), and her PK Token from localStorage. For the OSM header, the *alg* attribute is set to be the same as the *alg* attribute of the cic ("ECDSA" in our case). The *kid* attribute is set to be the SHA-3 hash of our PK Token, and the *typ* is set to "osm". We then sign Alice's messages and the generated header with her private key using `crypto.subtle.sign()`, and return the resulting OSM in JWT format.

4. **OSM Verification** On another end of the chatroom, Bob receives Alice's message as well as her PK Token. He wants to check if it's really Alice who's sending the message, so he verifies Alice's OSM with her PK Token using `verifyOSM()`. Bob first checks that all the header elements in her OSM are as expected (*typ*=="OSM", *alg* matches *cic*'s *alg*, and *kid*===`SHA-3(PK Token)`). After that, Bob checks that the PK Token is valid (see following section). Finally, Bob imports Alice's public key from the cic of her PK Token, and checks that the signature on the OSM verifies under Alice's key (using `crypto.subtle.verify()`). If all these checks pass, Bob knows that the message is indeed sent by Alice with her MIT identity.

5. **PK Token Verification** To verify a PK Token, we first split it into it's constituent parts: `userHeader, opHeader, payload, opSig, and userSig`. In the payload, we check that the *aud* claim must include our Client ID, and that the *iss* claim must be as we expect ("https://oidc.mit.edu/" in our case). We then check that the ID Token verifies using standard OIDC ID Token verification. After this, we check that the nonce of our payload matches the hash of the cic in our `userHeader`. Then, we check that the cic contains the correct elements (*alg*, *rz*, *upk*). Finally, we check that the user's signature verifies under the public key included in the cic (*upk*) again using `crypto.subtle.verify()`.

3.3.2 Security and Design Choices

Since the OpenPubKey implementation is designed with the OIDC Client, it follows the same security guidelines as enumerated in 3.2.4. Notably, we use `Crypto.subtle` functions for generating keys and signing & verification with said keys. We chose ECDSA for generating the ephemeral signing key & public key pair for the user. ECDSA requires shorter keys compared to RSA for the same level of security, leading to better performance.

To store the user's keys and PK Token for a session, we save it in localStorage and wipe when the user logs out. We can keep the keys and PK Token saved in localStorage since

these are client-side features. Improving the security of this part is part of our future work.

3.3.3 Extensibility

Our OpenPubKey implementation is designed to be as reusable and as extensible as possible. Extra features, such as multifactor authentication cosigners, can be added easily to the existing codebase. Moreover, the main functionality can be used with other OpenID clients without too many changes either. This allows our work to be built off of in future development or other projects, aligning with our main project priorities.

3.4 Chatroom Example

Note: To access the chatroom, users must log in with their MIT credentials through the MIT OpenID Connect service.

In the authenticated chatroom, we demonstrate how PK Tokens can be used to verify whether data is coming from a trusted user or not, which is fundamental to authentication. In our case, the data takes the form of a text message, and the trusted user is an identity holding an MIT credential.

The implementation consists of a frontend and a backend. The frontend performs the heavy lifting of signing and verifying OpenPubKey Signed Messages (OSMs), while the backend stores all OSMs in a data structure. For more complex projects, a database may be used.

Once logged in using an MIT credential, the user can see their kerberos and MIT email. The client utilizes methods from our `pktoken` module to generate a PK Token `opkService.getPKToken`, sign messages `opkService.generateOSM`, and verify messages `opkService.verifyOSM`. The user submits a message in the chat, which automatically creates an OSM consisting of their message and a signature of their message using their PK Token. To emphasize the decentralized nature of trust that can be achieved with OpenPubKey, our authenticated chatroom allows any user to verify any OSM at any time. All initial messages are displayed as unverified (grey check), and users can click on the check mark to verify them. The verification can either be accepted (green check) or rejected (red exclamation).

3.5 Other possible OpenPubKey usages

1. **Committing bets**: Two friends make a bet on something, and they want to prove they made it with their commitment. Say later one person tries to say they never made this bet, but because they signed their message (which is stored on the server), we have irrefutable proof that they did make the bet.

2. **Code signing**: Have a trusted verifier to have mapping of ID Tokens to public keys, and then whenever a user signs a commit they made, they can send it to the verifier and have it put into an append-only log.

3. **SSH**: SSH keys are difficult to manage. Instead, have users sign in through Google (or some other OpenID provider), and then have a PK Token that can act as their public SSH key.

4 Future Work

While we were able to achieve many of the original goals set out for this project, we recognize that there are aspects which could be further improved and developed on for our client framework.

1. **Security analysis of our code**: We recognize that code is rarely bug-free and secure the first time it is written. Before publishing our code to the MIT community, we would like to have it reviewed by other students, and members of the MIT OIDC team, for potential bugs and security issues.

2. **Advertisement to students**: A big motivation for us in creating this project is that we want for MIT students to actually be able to use this code when creating their web services. As a result, we plan to advertise our project through dormspam and get in contact with professors + TA's teaching web development class like web.lab to see they can mention it to their students. We also want to get feedback on how we can best enable the adoption of our authentication framework into existing services (rather than just new ones).

3. **Communicate server timeout issues to MIT OIDC**: As part of our initial survey and design work, we discover that one of the pain points reported by students when using the OIDC authentication service is that the login endpoint tends to have long delays at random times of the day (to the point of causing timeouts). We also encountered this issue multiple times in the development of this project. As a result, we plan to follow up with the MITRE team who created the MIT OIDC service to get their help on resolving this issue.

4. **Add session management for users**: One quality-of-life feature we would like to add to our OIDC client is the ability to create and manage sessions for logged in users. Currently, our system provides a way of authenticating users to the application, but it doesn't offer a generic way of tracking/authenticating actions done by that user after a successful login, outside of the OpenPubKey OSM messages. We would like to add a secure session management scheme, likely using libraries like 'express-session' or 'cookie-session'.

5. **Add MFA cosigners and better expiry to OpenPubKey implementation**: More improvements could be made to the OpenPubKey implementation of our project. One major features that OpenPubKey includes is cosigners for multifactor authentication, but the timeline of the project meant that it was not something we were able to

implement. Beyond this, our current expiry protocol clears keys every session, but the PK Token itself has no expiry. We know that this is not secure, and plan to implement a more secure expiry protocol, such as expiring tokens after 2 weeks or basing expiry off of the ID Token expiry. As part of our future work, we'd like to implement these features.

5 Conclusions

We were able to achieve our project's goal of building a simple and secure client integration for MIT student developers to quickly get authentication in their web services throughout MIT OIDC service. Along with this implementation, we are also providing student developers with the option to extend the end-user's Client ID Token into a secure signing key using OpenPubKey, allowing for signing of data in the web service. Additionally, we built an authenticated chatroom to better realize and understand the usage of our extension.

6 Acknowledgments

We want to thank Ethan Heilman from BastionZero for his valuable insights into OpenPubkey. We want to also thank the 6.857 course staff for their support and invaluable teachings throughout this entire semester.

References

[1] authts, oidc-client-ts, https://github.com/authts/oidc-client-ts.

[2] BastionZero, Security model, https://www.bastionzero.com/security-model.

[3] BastionZero, Webshell common ts, https://github.com/bastionzero/webshell-common-ts.

[4] E. Heilman, L. Mugnier, A. Filippidis, S. Goldberg, S. Lipman, Y. Marcus, and C. Unrein, Openpubkey: Augmenting openid connect with user held signing keys, https://eprint.iacr.org/2023/123, 2023.

[5] MIT Information Services and Technology, Authentication tools at mit, http://kb.mit.edu/confluence/display/istcontrib/Authentication+Tools+at+MIT.

[6] MIT Information Services and Technology, Logging in users to your application using openid connect, https://kb.mit.edu/confluence/display/istcontrib/Logging+in+Users+to+your+application+using+OpenID+Connect.

[7] robertvunabandi, passport-mitopenid, https://github.com/robertvunabandi/passport-mitopenid.

[8] D. N. Sakimura, J. Bradley, P. Identity, M. Jones, B. de Medeiros, and C. Mortimore, Openid connect implicit client implementer's guide 1.0-draft 42, Technical report.

[9] The MITRE Corporation and MIT KIT, Mit openid connect pilot - welcome, https://oidc.mit.edu/, 2014.