# Bethe OPS Project

## Developer Instructions: Login

**May 16, 2019**

# Table of Contents

# 1 Overview

This document describes how login authentication for the Bethe OPS system is set up and implemented.

The login for this system uses Shibboleth and Cornell's CUWebLogin. The code for the authentication is all in *app.js* and makes use of the Node.js *Passport* and *Passport-SAML* middleware.

# 2 Required Technologies

The required technologies for login authentication are:

- Node.js
- Shibboleth
- Passport
- SAML

# 3 Shibboleth and CUWebLogin Introduction

## 3.1 Shibboleth and CUWebLogin

Shibboleth is an open-source software that is often used for authentication for web-enabled based services. Many different organizations use it as it provides Single SignOn (SSO) within the same organization. Cornell is no exception as CUWebLogin shares SSO with Shibboleth so by integrating Shibboleth into Bethe OPS, the system can utilize CUWebLogin.

More information about Cornell's use of Shibboleth can be found at the following links:

- https://identity.cit.cornell.edu/authn/shibbolethintro.html
- https://confluence.cornell.edu/display/SHIBBOLETH/Shibboleth+at+Cornell+Page

## 3.2 Service Providers (SP), Identity Providers (IdP), and SAML

Service Providers are the entities/systems that require authentication while identity providers are the entities that provide the authentication service. They communicate by sending SAML (Security Assertion Markup Language) authentication requests and responses that are encrypted with certificates and keys.

Bethe OPS would be a service provider and Cornell would be the IdP.

### 3.3 SP and IdP Metadata

Both service providers and identity providers have information known as metadata. This metadata is stored in XML files and contains information about the information of the system. For example, for the Cornell IdP, the metadata can be found at the following link: https://shibidp.cit.cornell.edu/idp/shibboleth**.** As shown in the XML file, there is information about the X509 certificates used to encrypt authentication requests/responses, different URLs for sign on services, etc.

Similarly, service providers have their own metadata. In order for service providers to be fully integrated with an IdP, the IdP needs to have the SP metadata and validate it.

### 3.4 Certificates and Private Keys

In order for authentication requests to be sent between the SP and IdP securely, both SPs and IdPs make use of certificates and private keys. It is important to know that a certificate and private key form a corresponding pair. For example, an IdP has a certificate and private key that is used for encrypting and decrypting messages. The certificate is made public and is used by SPs to encrypt authentication requests to the IdP. The IdP, upon receiving such messages, uses its private key to decrypt the authentication request. A similar process is done for SPs.

### 3.5 Test and Production Versions of the IdP

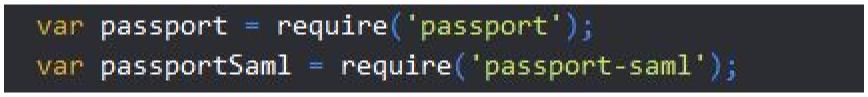
As mentioned above, in order for an application to make full use of an IdP, the SP metadata needs to be created for the application and then verified before the IdP can integrate the new application into its system. Since this can take a while, identity providers offer "test" versions of their services which applications can use to test authentication without creating SP metadata.

# 4 Login for Bethe-OPS with the test IdP

The information below is for getting the system to work with the test IdP Cornell offers.

### 4.1 Passport and Passport-SAML

*Passport* is a widely popular middleware of Node.js used for authentication. It has many different forms for different kinds of authentication. For example, some version of *Passport* offer authentication with gmail while others use facebook. Since the desired login for the system makes use of Shibboleth and SAML, the extension of *Passport* is the *Passport-SAML.* These middleware modules are imported into *app.js* as shown below.

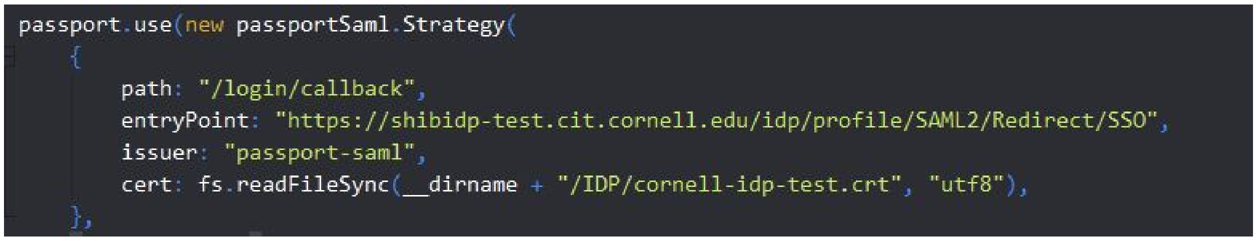


```
var passport = require('passport');
var passportSaml = require('passport-saml');
```

There are a few key functions in *Passport* that define how the authentication for the system will work. These are described below.

## 4.2 Passport's use( ) and PassportSAML's strategy( )

To use *Passport,* first define what type of "strategy" it will use for authentication. These "strategies" define parameters that will be used for routing and authentication. Since Shibboleth used, the strategy defined by *Passport-SAML* is used. The code for defining the strategy is below.



```
passport.use(new passportSaml.Strategy(
    {
        path: "/login/callback",
        entryPoint: "https://shibidp-test.cit.cornell.edu/idp/profile/SAML2/Redirect/SSO",
        issuer: "passport-saml",
        cert: fs.readFileSync(__dirname + "/IDP/cornell-idp-test.crt", "utf8"),
    },
```

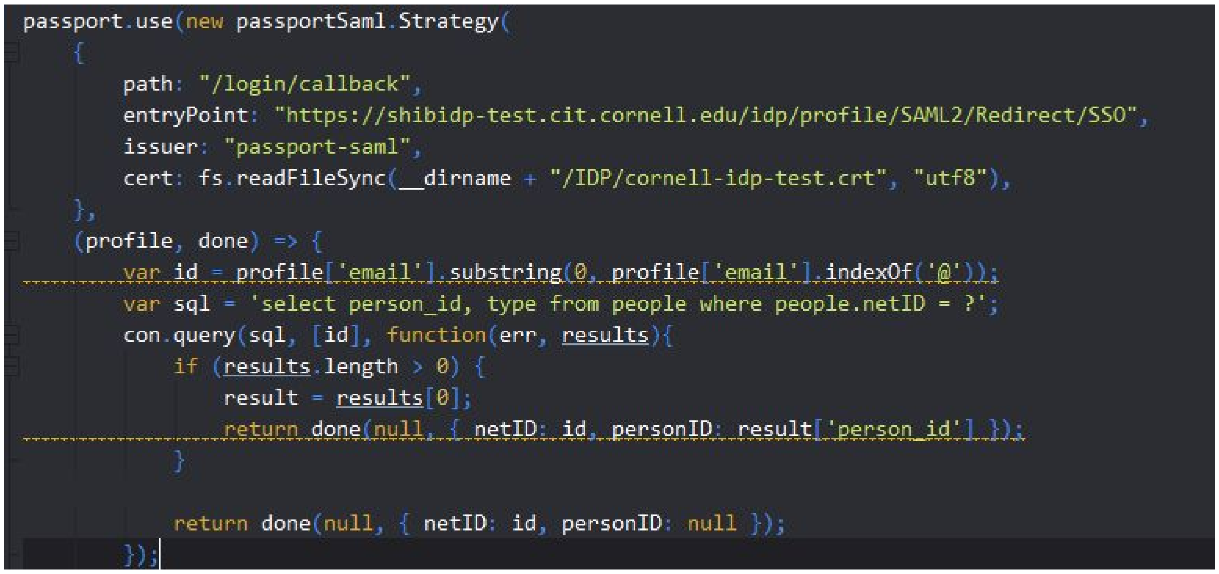
As shown above, the strategy in *Passport-SAML* requires a few pieces of information as follows:

- *path*: the callback route that will be loaded after the IdP does authentication and returns back to the website
- *entryPoint*: the IdP entrypoint. In other words, it is the URL to which the application should direct authentication requests
- *issuer*: information that informs the IdP of who is sending the authentication requests; since *Passport-SAML* is used, simply set it as *passport-saml*
- *cert*: the IdP's public signing certificate used to validate signatures of incoming SAML requests

The entrypoint and certificate was found by going to Cornell's confluence page on Shibboleth here. This page contains the metadata, entity IDs, and certificates for both the test and production IdPs.

Since only the test IdP is used, there is no need for SP metadata and for the application to sign the SAML requests it sends. If the production version of the IdP is used, there would be other pieces of information required. This will be discussed in a later section.

If the authentication is successful on the IdP side, the IdP sends back a SAML response containing information about the user. This information can be handled in a callback function in *passport.use* as shown below in the full *passport.use* function.



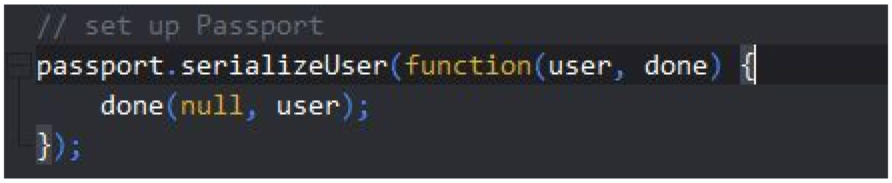
```
passport.use(new passportSaml.Strategy(
    {
        path: "/login/callback",
        entryPoint: "https://shibidp-test.cit.cornell.edu/idp/profile/SAML2/Redirect/SSO",
        issuer: "passport-saml",
        cert: fs.readFileSync(__dirname + "/IDP/cornell-idp-test.crt", "utf8"),
    },
    (profile, done) => {
        var id = profile['email'].substring(0, profile['email'].indexOf('@'));
        var sql = 'select person_id, type from people where people.netID = ?';
        con.query(sql, [id], function(err, results){
            if (results.length > 0) {
                result = results[0];
                return done(null, { netID: id, personID: result['person_id'] });
            }

            return done(null, { netID: id, personID: null });
        });
```

The IdP sends back information, which can be accessed in the variable *profile*. This information can be stored and utilized in any way that is needed. Thus, in the code above, the email of the authenticated user is retrieved from the profile. Since the system deals with Cornell emails, the net ID can be obtained by getting the characters before the '@' symbol. The net ID is then used as a key into the database to check whether this user exists in the system's database or not. The person's net ID and person ID, which is the person's identifier in the database (this will be null if the user does not exist), are stored in an object, and the function, *done*, is invoked. *done* passes this information to another Passport function called *passport.serialize.*

## 4.3 Passport.serializeUser

*passport.serializeUser* takes the objective from *passport.use* and puts it into a cookie. This cookie will be used by two other functions described next.



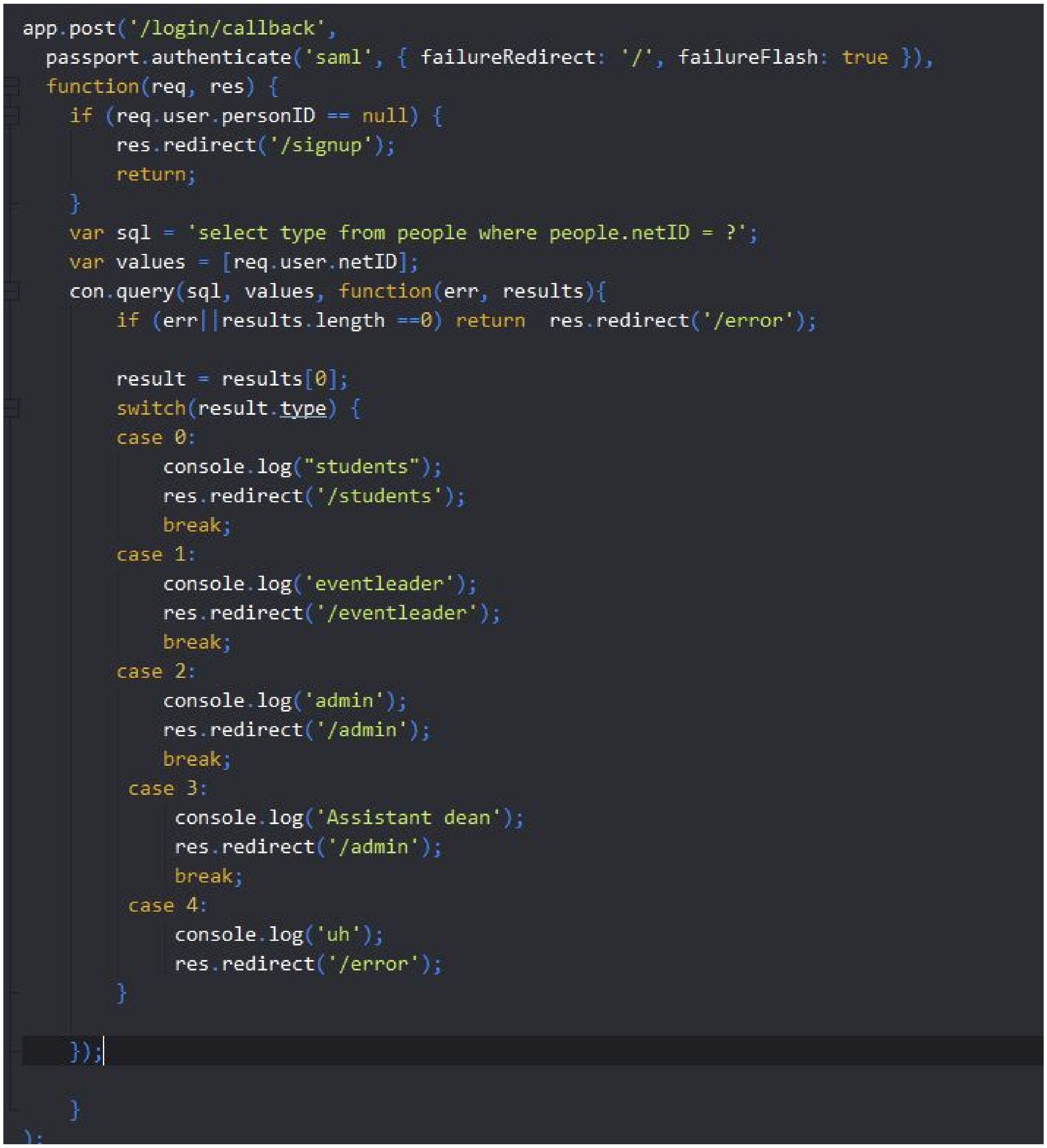
```
// set up Passport
passport.serializeUser(function(user, done) {
    done(null, user);
});
```

## 4.4 Passport.deserializeUser

Every time a user logs into the website or access a page, the deserialize function is called. This function does the opposite of deserializeUser in that it takes in the cookie and allows the information to be used or passed to the rest of the system.

## 4.5 app.post('/login/callback')

As mentioned above, *app.post('login/callback')* is the route that is loaded by the application once the IdP sends back its authentication response. While serializeUser( ) and deserializeUser( ) are for handling the data from the IdP and storing them or retrieving them from cookies, this route handles where to redirect the web application once the user has been authenticated. The code is shown below.



```
app.post('/login/callback',
  passport.authenticate('saml', { failureRedirect: '/', failureFlash: true }),
  function(req, res) {
    if (req.user.personID == null) {
        res.redirect('/signup');
        return;
    }
    var sql = 'select type from people where people.netID = ?';
    var values = [req.user.netID];
    con.query(sql, values, function(err, results){
        if (err||results.length ==0) return  res.redirect('/error');

        result = results[0];
        switch(result.type) {
        case 0:
            console.log("students");
            res.redirect('/students');
            break;
        case 1:
            console.log('eventleader');
            res.redirect('/eventleader');
            break;
        case 2:
            console.log('admin');
            res.redirect('/admin');
            break;
         case 3:
             console.log('Assistant dean');
             res.redirect('/admin');
             break;
         case 4:
             console.log('uh');
             res.redirect('/error');
        }

    });

    }
);
```

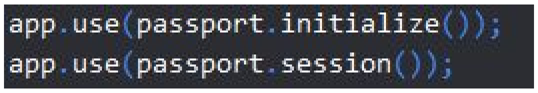
Since the system has different types of users (students, event leaders, and admins), the system needs to re-route the user to different home pages depending on what type of privilege (s)he has.

First, if the *personID* field in the cookie is null, the user is redirected to a sign-up page to register as a student. This is as the *personID* field is null if the logged in user was not found in the database.

If the *personID* field is not null, then the person ID is used to query into the database and find the type/privilege level of the user. Based on his/her privilege level, the user is redirected to the student, event leader, or admin home page.

### 4.6 Initializing passport

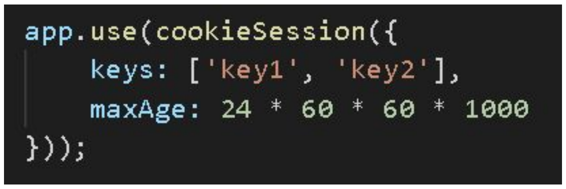
To initialize the passport middleware and session, the Express middleware function, *app.use*, is called on *passport* as in the following code.



```
app.use(passport.initialize());
app.use(passport.session());
```

## 5 Encrypting the Cookies with Cookie-Session

As mentioned above, *passport.serializeUser* stores information about the user returned from the IdP in a cookie. However, this cookie is by itself not encrypted. For best practice, it is best to encrypt the cookie to avoid any leaks of information. To do this, the middleware, *cookie-session*, is used. The following code shows initialization of *cookie-session.*



```
app.use(cookieSession({
    keys: ['key1', 'key2'],
    maxAge: 24 * 60 * 60 * 1000
}));
```

The *keys* field is simply a list of strings that are used to encrypt the cookies. The *maxAge* field refers to how long the cookies will be valid without being set again. The time is in milliseconds so in the image, cookies are set to be valid for up to a day.

By calling *app.use* on *cookieSession*, the encryption is automatically handled.

## 6 Switching the Test IdP to the Production IdP

Due to time constraints, the current system's login authentication system makes use of the test IdP. However, if the system is to be rolled out into a production version, it is best practice to switch to using the production IdP. Several things are needed in order to accomplish this.

## 6.1 Using the Production IdP Certificate

Currently, the certificate passed into the *cert* field in the strategy is the certificate of the test IdP which is stored in the IdP folder of the project. This needs to be changed to the production IdP certificate, which can be found here.

## 6.2 Generate (Public Certificate, Private Key Pair) Using Tools such as OpenSSL

This certificate will be made public and be used to encrypt SAML messages sent to Bethe OPS. The private key will then be used to decrypt those messages.

## 6.3 Generate SP Metadata

There are two ways to do this. One is based on an example found online that also uses Passport-SAML. The other uses an online tool called SAMLTool.

### 6.3.1 Use Passport-SAML's generateServiceProviderMetadata( )

The documentation on this function can be found on the Github repository for *Passport-SAML*.

An example of how to set up the strategy and generate SP metadata is given here. As shown in this example, the strategy has many more fields. The *callbackUrl, entryPoint,* and *issuer* are the same. The *cert* should use the IdP production certificate. The most important new fields are *decryptionPvK* and *privateCert*. Both of these point to the private key generated in part 6.2 above.

Finally, to create the SP metadata, call the function, generateServiceProviderMetadata( ), on the strategy object. Currently, the strategy is defined inside the *passport.use* function so to get this working, the strategy should be defined outside *passport.use* so it can be both passed into *passport.use* as well as used to call the SP generating function. According to the example and documentation, the public certificate should be passed into this function and the output is the PS metadata.

### 6.3.2 Use SAMLTool to Create the SP Metadata

If the above method does not work, Cornell IT recommends SAMLTool. SAMLTool is an online source that provides tools for creating SP metadata. To do this, there are two steps.

First, an X.509 certificate needs to be created. The tool to do this can be found here. This tool takes in information about the application as well as the private key and certificate of the SP to create a self-signed X.509 certificate.

Next, the X.509 certificate is used to create the SP metadata. The tool to do this can be found here. This tool requires information about the entity ID, endpoints or login routes of the Bethe OPS system, and the public X.509 certificate to create the SP metadata. Since this is an external tool, it is recommended that the method presented in section 6.3.1 be tried first since it uses functions defined by *Passport* and *Passport-SAML*.

## 6.4 Register the SP with the IdP

Once the SP metadata is generated, it needs to be sent to Cornell IT for formal verification. If it passed their tests, the SP for the Bethe OPS system is registered and login authentication will be able to use the production IdP instead of the test IdP.