tangentsky Documentation

Release 0.2.1

Josh Bialkowski

Nov 02, 2019

Contents

Simple oauth2 subrequest handler for reverse proxy configurations

Installation

1.1 Install with pip

The easiest way to install `oauthsub` is from pypi.org using [pip](pip). For example:

```
pip install oauthsub
```

If you're on a linux-type system (such as ubuntu) the above command might not work if it would install into a system-wide location. If that's what you really want you might need to use `sudo`, e.g.:

```
sudo pip install oauthsub
```

In general though I wouldn't really recommend doing that though since things can get pretty messy between your system python distributions and your `pip` managed directories. Alternatively you can install it for your user with:

```
pip install --user oauthsub
```

which I would probably recommend for most users.

1.2 Install from source

You can also install from source with pip. You can download a release package from github and then install it directly with pip. For example:

```
pip install v0.1.0.tar.gz
```

Note that the release packages are automatically generated from git tags which are the same commit used to generate the corresponding version package on `pypi.org`. So whether you install a particular version from github or pypi shouldn't matter.

Pip can also install directly from github. For example:

```
pip install git+https://github.com/cheshirekow/oauthsub.git
```

If you wish to test a pre-release or dev package from a branch called `foobar` you can install it with:

```
pip install "git+https://github.com/cheshirekow/oauthsub.git@foobar"
```

Usage

```
usage: oauthsub [-h] [--dump-config] [-v] [-l {debug,info,warning,error}]
                [-c CONFIG_FILE] [-s {flask,gevent,twisted}]
                [--rooturl ROOTURL] [--flask-debug [FLASK_DEBUG]]
                [--flask-privkey FLASK_PRIVKEY]
                [--response-header RESPONSE_HEADER]
                [--allowed-domains [ALLOWED_DOMAINS [ALLOWED_DOMAINS ...]]]
                [--host HOST] [--port PORT] [--logdir LOGDIR]
                [--route-prefix ROUTE_PREFIX]
                [--session-key-prefix SESSION_KEY_PREFIX]
                [--bypass-key BYPASS_KEY] [--custom-template CUSTOM_TEMPLATE]
                [--enable-forbidden [ENABLE_FORBIDDEN]]

This lightweight web service performs authentication. All requests that reach
this service should be proxied through nginx. See:
https://developers.google.com/api-client-library/python/auth/web-app

optional arguments:
  -h, --help            show this help message and exit
  --dump-config         Dump configuration and exit
  -v, --version         show program's version number and exit
  -l {debug,info,warning,error}, --log-level {debug,info,warning,error}
                        Increase log level to include info/debug
  -c CONFIG_FILE, --config-file CONFIG_FILE
                        use a configuration file
  -s {flask,gevent,twisted}, --server {flask,gevent,twisted}
                        Which WGSI server to use
  --rooturl ROOTURL     The root URL for browser redirects
  --flask-debug [FLASK_DEBUG]
                        Enable flask debugging for testing
  --flask-privkey FLASK_PRIVKEY
                        Secret key used to sign cookies
  --response-header RESPONSE_HEADER
                        If specified, the authenticated user's ``username``
                        will be passed as a response header with this key.
```

```
--allowed-domains [ALLOWED_DOMAINS [ALLOWED_DOMAINS ...]]
                     List of domains that we allow in the `hd` field of
                     thegoogle response. Set this to your company gsuite
                     domains.
--host HOST          The address to listening on
--port PORT          The port to listen on
--logdir LOGDIR      Directory where we store resource files
--route-prefix ROUTE_PREFIX
                     All flask routes (endpoints) are prefixed with this
--session-key-prefix SESSION_KEY_PREFIX
                     All session keys are prefixed with this
--bypass-key BYPASS_KEY
                     Secret string which can be used to bypass
                     authorization if provided in an HTTP header
                     `X-OAuthSub-Bypass`
--custom-template CUSTOM_TEMPLATE
                     Path to custom jinja template
--enable-forbidden [ENABLE_FORBIDDEN]
                     If true, enables the /forbidden endpoint, to which you
                     can redirect 401 errors from your reverse proxy. This
                     page is a simple message with active template but
                     includes login links that will redirect back to the
                     forbidden page after a successful auth.
```

2.1 Configuration

oauthsub is configurable through a configuration file in python (the file is exec``ed). Each configuration variable can also be specified on the command line (use ``oauthsub --help to see a list of options). If you'd like to dump a configuration file containing default values use:

```
oauthsub --dump-config
```

Which outputs something like:

```
.. dynamic: config-begin
```

```python
# The root URL for browser redirects
rooturl = 'http://localhost'

# Enable flask debugging for testing
flask_debug = False

# Secret key used to sign cookies
flask_privkey = 'KALJE0Unas2dd8ao3p/T55htwbL5RrKX'

# If specified, the authenticated user's ``username`` will be passed as a
# response header with this key.
response_header = None

# List of domains that we allow in the `hd` field of thegoogle response. Set
# this to your company gsuite domains.
allowed_domains = ['gmail.com']
```

```python
# The address to listening on
host = '0.0.0.0'

# The port to listen on
port = 8081

# Directory where we store resource files
logdir = '/tmp/oauthsub/logs'

# Flask configuration options. Set session config here.
flaskopt = {
  "PERMANENT_SESSION_LIFETIME": 864000,
  "SESSION_FILE_DIR": "/tmp/oauthsub/session_data",
  "SESSION_TYPE": "filesystem"
}

# All flask routes (endpoints) are prefixed with this
route_prefix = '/auth'

# All session keys are prefixed with this
session_key_prefix = 'oauthsub-'

# Secret string which can be used to bypass authorization if provided in an HTTP
# header `X-OAuthSub-Bypass`
bypass_key = None

# Dictionary mapping oauth privider names to the client secrets for that
# provider.
client_secrets = {}

# Path to custom jinja template
custom_template = None

# If true, enables the /forbidden endpoint, to which you can redirect 401 errors
# from your reverse proxy. This page is a simple message  with active template
# but includes login links that will redirect back to the forbidden page after a
# successful auth.
enable_forbidden = True

# Which WGSI server to use (flask, gevent, twisted)
server = 'flask'



# This is not used internally, but is used to implement our user lookup
# callback below
_user_map = {
    "alice@example.com": "alice",
    "bob@example.com": "bob"
}

# This is a callback used to lookup the user identity based on the credentials
# provided by the authenticator.
def user_lookup(authenticator, parsed_response):
  if authenticator.type == "GOOGLE":
    # Could also use `id` to lookup based on google user id
    return _user_map.get(parsed_response.get("email"))
```

```
    return None
```

Examples

This section will demonstrate how to use `oauthsub` to secure various services behind NGINX as a reverse proxy, with authentication provided by one of many popular providers. The basic setup for each service is shown in figure Fig. 3.1.

Fig. 3.1: The basic authentication setup.

Before proceeding with these examples, you'll need to setup at least one `oauth2` provider. See the tutorials below on how to do that. Then you can create a basic nginx configuration before finally configuring your backend service integration.

3.1 Providers

In order to take advantage of a public `oauth2` provider you must do some setup. This usually involves registering an application and getting an API token (a `client_secret`). This section will walk you through the process of registering with various popular providers.

In order to be explicit, we'll configure the client information for the test setup that we've configured:
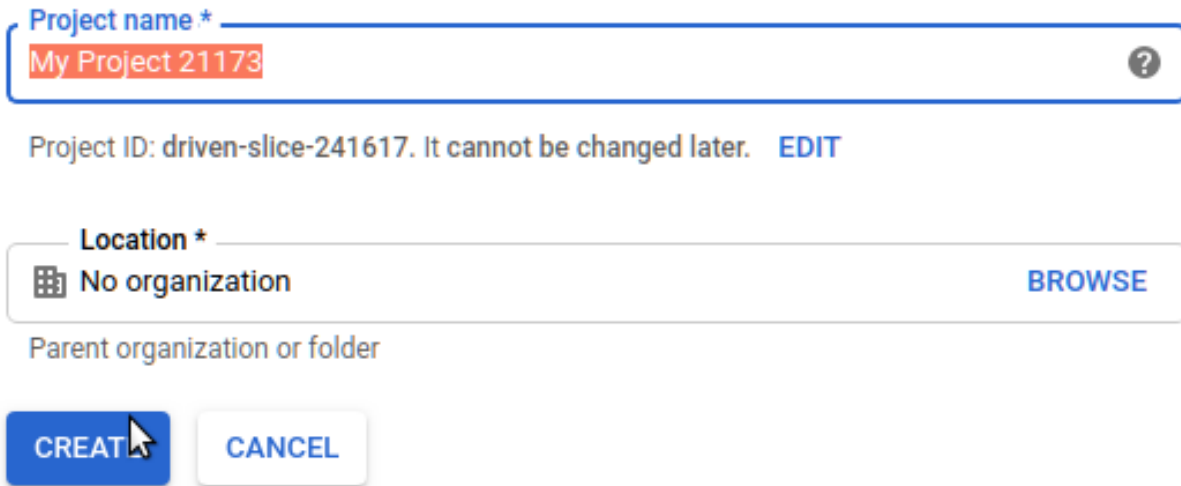
- `nginx` on `localhost:8080`
- `oauthsub` on `localhost:8081`

You will need to adapt these instructions for your production setup.

3.1.1 Google

Go to the Google Developer Dashboard and create a new project.

Give it a name.

Select the project in the top left next to the GoogleAPIs logo. Click "Credentials" under the menu on the left of the screen.

Click "Create credentials":

Select "Web Application". Give it a name. Then fill in the Javascript origins:

- http://lvh.me:8080/

- http://lvh.me:8081/

- https://lvh.me:8443/

and authorized redirects:

- http://lvh.me:8080/auth/callback?provider=google

- http://lvh.me:8081/auth/callback?provider=google

- https://lvh.me:8443/auth/callback?provider=google

Then click "create" and copy out the client id and secret from the popup.

The client id is also available on the dashboard after you create it:

Note: We add three authorized domains for different testing scenarios. The 8081 port will be the raw service. 8080 will be an nginx reverse proxy listening over unencrypted http. 8443 will be an nginx reverse proxy listening over encrypted https (with a self-signed certificate).

Note: As of January 2019 Google has recently changed their developer settings and requirements for OAUTH access. They used to allow *localhost* and now they do not. An alternative is to use *lvh.me* which currently resolves through DNS to 127.0.0.1. Be careful, however, as this is a common solution cited on the interwebs but no one seems to know who controls this domain and they may be nefarious actors.

For deployment you'll want to add a redirect like this:

← Create OAuth client ID

For applications that use the OAuth 2.0 protocol to call Google APIs, you can use an OAuth 2.0 client ID to generate an access token. The token contains a unique identifier. See Setting up OAuth 2.0 for more information.

Application type
- ⦿ Web application
- ○ Android Learn more
- ○ Chrome App Learn more
- ○ iOS Learn more
- ○ Other

Name ⑦

oauthsub-client

Restrictions

Enter JavaScript origins, redirect URIs, or both Learn More

Origins and redirect domains must be added to the list of Authorized Domains in the OAuth consent settings.

Authorized JavaScript origins

For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (https://*.example.com) or a path (https://example.com/subdir). If you're using a nonstandard port, you must include it in the origin URI.

http://lvh.me:8080	🗑
http://lvh.me:8081	🗑
https://lvh.me:8443	🗑

https://www.example.com

Type in the domain and press Enter to add it

Authorized redirect URIs

For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

http://lvh.me:8080/auth/callback?provider=google	🗑
http://lvh.me:8081/auth/callback?provider=google	🗑
https://lvh.me:8443/auth/callback?provider=google	🗑

https://www.example.com

Type in the domain and press Enter to add it

Create Cancel

OAuth client

The client ID and secret can always be accessed from Credentials in APIs & Services

> ⓘ OAuth is limited to 100 sensitive scope logins until the OAuth consent screen is published. This may require a verification process that can take several days.

Here is your client ID

Here is your client secret

OK

Credentials   OAuth consent screen   Domain verification

Create credentials ▾   Delete

Create credentials to access your enabled APIs. For more information, see the authentication documentation.

OAuth 2.0 client IDs

	Name	Creation date ∨	Type	Client ID		
☐	oauthsub-client	May 24, 2019	Web application			

```
https://server.yoursite.com/auth/callback?provider=google
```

3.1.2 Github

Go to the Github Developer Settings.
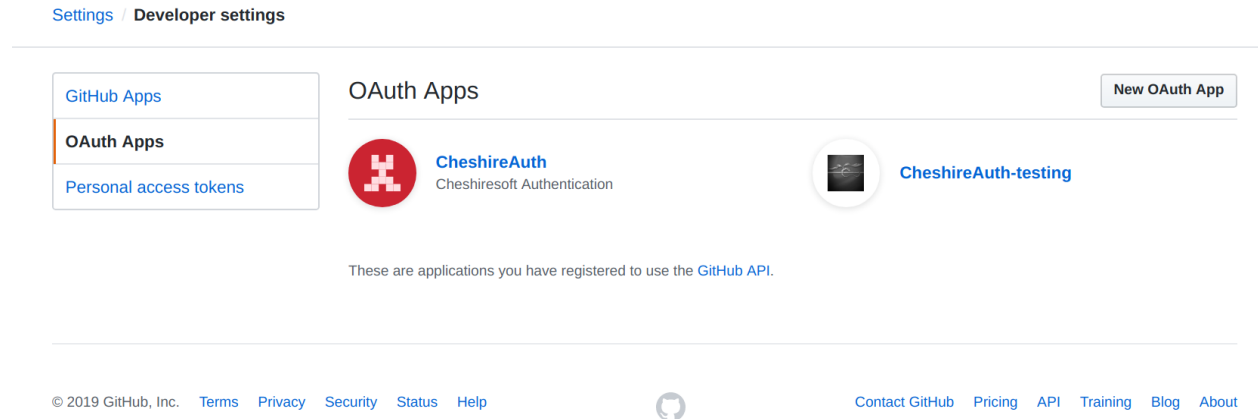
Click "New OAuth App". Fill out the form. Set the "Authorization Callback URL" to:

```
http://lvh.me:8080/auth/callback?provider=github
```

for testing, or your real server for deployment. Note that, unlike google, we do not need to use the full exact URL (in particular, we can leave off the `?provider=` bits.

> **Warning:** Github does not allow you to authorize multiple redirects for an application. If you wish to test multiple configurations, you will need to update the Authorized Callback URL each time, or register multiple applications.

Copy down the "Client ID" and "Client Secret" and add them to your `config.py`.

3.1.3 Microsoft

Go to the Azure Active Directory admin center.

Click "Azure Active Directory" on the left, and then "App registrations". On the top of the page click "New registration".

Give it a name and set the redirect uri to "http://localhost:8080/auth/callback". Then click "Register" at the bottom.

> **Note:** Microsoft does not allow `http://` for anything other than localhost, so we can't use `http://lvh.me:8080` like we can with the other providers.

On the next page copy off the "Application (client) ID".

Then click "Certificates & secrets" and click "New client secret".

Give it a name, set an expiration, then click "Add".

Then copy the newly created client secret.

Register a new OAuth application

Application name *

deleteme

Something users will recognize and trust.

Homepage URL *

https://github.com/cheshirekow

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

http://lvh.me:8080/auth/callback

Your application's callback URL. Read our OAuth documentation for more information.

Register application   **Cancel**

Client ID
0cf93ced4e5ad8f60e13

Client Secret
dfeb471d346c0c2967b35b9c8c802abda690698d

[**Revoke all user tokens**]   [**Reset client secret**]

* Name

The user-facing display name for this application (this can be changed later).

| oauthsub-client | ✓ |

Supported account types

Who can use this application or access this API?

◯ Accounts in any organizational directory

⦿ Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

Help me choose...

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

| Web ⌄ | http://localhost:8080/auth/callback | ✓ |

By proceeding, you agree to the Microsoft Platform Policies ⧉

[Register]

3.1.4 Atlassian

First, go to the Atlassian Apps management page. Click "Create new App":
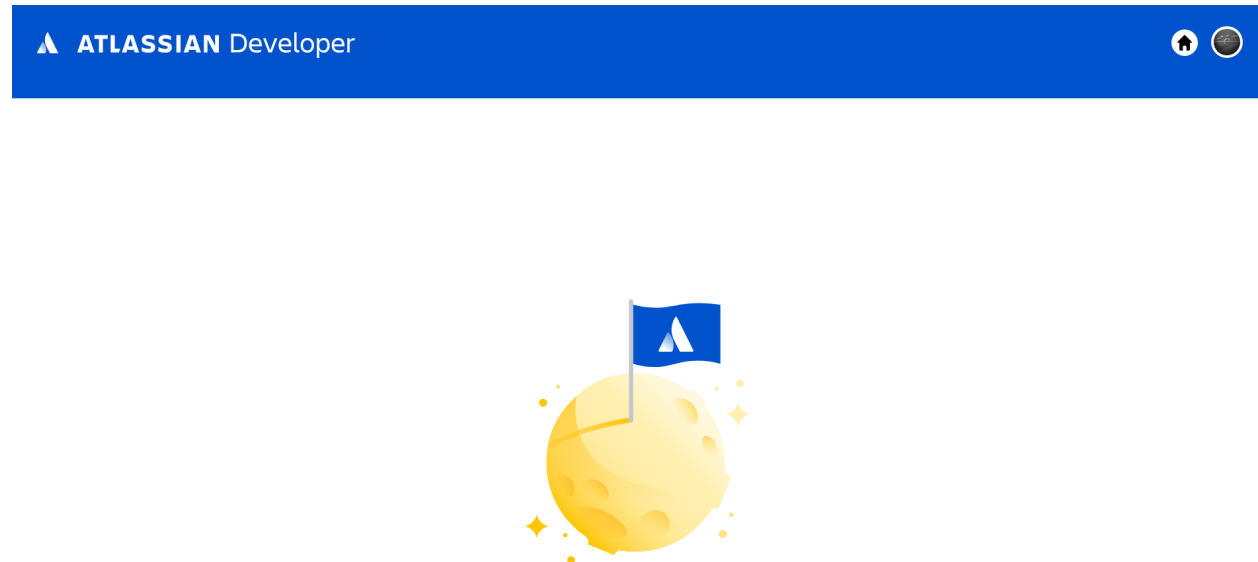
Give your app a name, agree to terms, then click "Create":

On the app info page, copy out the client id and client secret. Then click "OAuth 2.0 (3LO)" on the left panel.

Enter "https://lvh.me:443/auth/callback?provider=atlassian" in the field and click "Save changes".

Note: Like github, you can only enter one callback, so you'll need a separate app for testing purposes.

Note: Atlassian will not allow you to register an *http://* URL. . . even for testing so you'll need something listening with https.

Documentation coming soon. See also the atlassian page.

3.2 NGINX configuration

Here is an nginx configuration that should illustrate the foundation of working with `oauthsub`. See the comments inline for additional information.

Create a new app

An app provides API credentials for Atlassian products and services, as well as features such as OAuth 2.0 (3LO).

Name *

CheshireAuth

Name your app according to its purpose, for example, Dropbox integration or Timesheets for Jira.

☑ I agree to be bound by Atlassian's developer terms.

Create   Cancel

App details

C

Change avatar

Name *

CheshireAuth

Description

Tell customers what your app does

Your app description will appear in the user's app directory

Save changes

Client ID

ccKxvDhrNdLczKo9ML1FWCanEkBsY   Copy

Secret

••••••••••••••••••••••••••••••••••   Copy   •••

Use the Client ID and Secret for authentication. See the OAuth 2.0 (3LO) guide to learn more.

OAuth 2.0 authorization code grants (3LO) for apps

Configure OAuth 2.0 authorization code grants to allow your app to access data (within specific scopes) from Atlassian APIs on the user's behalf. Learn more about OAuth 2.0 authorization code grants.

Callback URL *

https://lvh.me:8443/auth/callback

Save changes   Discard changes

3.2.1 Basic setup

The nginx server will serve anything under `public` or `auth` without authentication or authorization. For any other request, nginx will forward the http headers to the authentication service over http. The authentication service will return an HTTP status code of 200 if the user is authenticated/authorized, and 401 if they are not. All users with who login with an account that is within the authorized domain list is authorized.

The nginx server proxies all requests rooted at `auth/` to the authentication service which is a python flask application. The auth service uses a session (persisted through a cookie) to store the user's authenticated credentials (email address reported by google). If the user is not authenticated or is not authorized, the 401 error page is served by the authentication service to provide some info about why the request was denied (i.e. what they are currently logged in as). There is also a link on that page to login if they are not.

```
location / {
  # Use ngx_http_auth_request_module to auth the user, sending the
  # request to the /auth/query_auth URI which will return an http
  # error code of 200 if approved or 401 if denied.
  auth_request /auth/query_auth;

  # First attempt to serve request as file, then
  # as directory, then fall back to displaying a 404.
  try_files $uri $uri/ =404;
}

# Whether we have one or not, browsers are going to ask for this so we
# probably shouldn't plumb it through auth.
location = /favicon.ico {
  auth_request off;
  try_files $uri $uri/ =404;
}

# The authentication service exposes a few other endpoints, all starting
# with the uri prefix /auth. These endpoints are for the oauth2 login page,
# callback, logout, etc
location /auth {
  auth_request off;
  proxy_pass http://localhost:8081;
  proxy_pass_request_body on;
  proxy_set_header X-Original-URI $request_uri;
}

# the /auth/query URI is proxied to the authentication service, which will
# return an http code 200 if the user is authorized, or 401 if they are
# not
location = /auth/query_auth {
  proxy_pass http://localhost:8081;
  proxy_pass_request_body off;
  proxy_set_header Content-Length "";
  proxy_set_header X-Original-URI $request_uri;
  proxy_pass_header X-OAuthSub-Bypass-Key;
  proxy_pass_header X-OAuthSub-Bypass-User;
}

# if the server is using letsencrypt  certbot then we'll want this
# directory to be accessible publicly
location /.well-known {
  auth_request off;
}
```

(continues on next page)

```
# we may want to keep some uri's available without authentication
location /public {
  auth_request off;
}

# for 401 (not authorized) redirect to the auth service which will include
# the original URI in it's oauthflow and redirect back to the originally
# requested page after auth
error_page 401 /auth/forbidden;
```

3.2.2 Remote User Tokens

If you want `oauthsub` to forward the username through a header variable then set the `request_header` configuration variable for `oauthsub` and add the following to your nginx configuration. In this example the `request_header` is `X-User` and the protected service service listening on 8082.:

```
location / {
    auth_request       /auth/query_auth;
    auth_request_set $user $upstream_x_user;
    proxy_set_header x-user $user;
    proxy_pass       http://localhost:8082;
}
```

In this case the protected service will need to be configure to accept the username in the *X-User* request header.

> **Warning:** Pay particular attention to such protected services when making changes to your nginx configuration. If you remove the `auth_request` but don't change the underlying service configuration anyone will be able to spoof arbitrary user identities by simply providing the correct *X-User* header.

3.2.3 Testing the service

While doing development and testing it can be troublesome to edit system level configurations and start/stop root-owned services. You can run NGINX in the foreground as an unpriviledged user.

To execute in foreground add the following to the nginx config:

```
daemon off;
```

On an ubuntu system, for example, you can copy `/etc/nginx/nginx.conf` and then add `daemon off;` to the top. You can then embed your testing site configuration, in which case you will end up with a file like this

```
daemon off;
worker_processes auto;
pid /tmp/nginx.pid;

events {
  worker_connections 768;
}

http {
  sendfile on;
```

```
tcp_nopush on;
tcp_nodelay on;
keepalive_timeout 65;
types_hash_max_size 2048;
include /etc/nginx/mime.types;
default_type application/octet-stream;
ssl_protocols TLSv1 TLSv1.1 TLSv1.2; # Dropping SSLv3, ref: POODLE
ssl_prefer_server_ciphers on;
access_log /tmp/nginx-access.log;
error_log /tmp/nginx-error.log;
gzip on;
gzip_disable "msie6";

server {

  listen 8080 default_server;
  listen [::]:8080 default_server;

  index index.html index.htm index.nginx-debian.html;
  server_name cheshiresoft;
  root /tmp/webroot;

  location / {
    auth_request /auth/query_auth;
    try_files $uri $uri/ =404;
  }

  location = /favicon.ico {
    auth_request off;
    try_files $uri $uri/ =404;
  }

  location /auth {
    auth_request off;
    proxy_pass http://localhost:8081;
    proxy_pass_request_body on;
    proxy_set_header X-Original-URI $request_uri;
  }

  location = /auth/query_auth {
    proxy_pass http://localhost:8081;
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
    proxy_pass_header X-OAuthSub-Bypass-Key;
    proxy_pass_header X-OAuthSub-Bypass-User;
  }

  location /public {
    auth_request off;
  }

  error_page 401 /auth/forbidden;
  }
}
```

You can then run nginx as follows:

```
nginx -p <prefix> -c <prefix>/nginx.conf \
  -g "error_log <prefix>/nginx-error-log"
```

Note that the `-g "error_log...` part is required to work-around the fact that nginx tries to write the error log to a root-owned location even before reading in the configuration file.

3.2.4 Executing

Write your client secrets to `/tmp/config.py` and then start simple auth with:

```
oauthsub --flask-debug \
         --config /tmp/config.py \
         --port 8081 \
         --rooturl http://localhost:8080
```

Write the above configuration to `/tmp/nginx.conf` and start nginx with:

```
nginx -c /tmp/nginx.conf -g "error_log /tmp/nginx-error.log;"
```

And navigate to "http://localhost:8080/" with your browser. You should be initially denied, required to login, and then directed to the default "welcome to nginx" page (unless you've written something else to your default webroot).

3.3 Gerrit

Gerrit provides git-based code hosting and code review services. It can be configured to accept the Remote User Token from `oauthsub`. There are a few relevant sections of `gerrit.config`. First, with gerrit sitting behind a reverse proxy you must tell gerrit what it's URL is so that it can properly contruct links. For our testing configuration we'll use the following:

```
[gerrit]
  canonicalWebUrl = http://lvh.me:8080/gerrit/
```

Secondly, we need to tell gerrit which port to listen on for http connections. We'll setup gerrit to listen on 8082:

```
[httpd]
  listenUrl = http://*:8082/gerrit/
```

Note: For a production server, consider using `proxy-http://127.0.0.1:8082/gerrit/` instead of `http://`

Lastly, we need to tell gerrit to enable HTTP header authentication, and which header to look in. For our example setup, that gives us:

```
[auth]
  type = HTTP
  httpHeader = X-Gsuite-User
  emailFormat = {0}@example.com
```

And now that gerrit is configured, we need to update the nginx configuration to proxy it. Add the following to your nginx site configuration:

```
location = /gerrit {
  return 302 /gerrit/;
}

location /gerrit/ {
  auth_request /auth/query_auth;
  auth_request_set $user $upstream_http_x_gsuite_user;
  proxy_set_header X-Gsuite-User $user;

  proxy_pass http://localhost:8082;
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
}
```

Note that nginx behaves differently depending on whether or not the `proxy_pass` URL ends in a slash. Without the trailing slash, as we have done here, will forward the whole URI down to the proxied service. In this case that means that all requests that gerrit sees will be prefixed by the `gerrit/` path. As alternative configuration, we could configure nginx to forward only the relative URI (i.e. strip the `gerrit/` prefix) and then we would change the gerrit config to `listenUrl = http://*:8082/`.

3.4 Buildbot Master

Buildbot is a continuous integration framework in python. We can configure the master to run behind nginx and to consume Remote User Tokens from `oauthsub`.

In our example setup we will have buildbot listen on port 8083. In your buildbot master configuration (`master.cfg`) add the following:

```
c['www'] = {
    "port": 8083,
    "plugins": {
        "waterfall_view": {},
        "console_view": {},
        "grid_view": {},
    },
    "auth": util.RemoteUserAuth(
        header="X-Gsuite-User",
        headerRegex=r"(?P<username>[^ @]+)@?(?P<realm>[^ @]+)?"),
}
```

Then in your nginx configuration:

```
location = /buildbot {
  return 302 /buildbot/;
}

location /buildbot/ {
  auth_request /auth/query_auth;
  auth_request_set $user $upstream_http_x_gsuite_user;
  proxy_set_header X-Gsuite-User $user;

  proxy_pass http://localhost:8083/;
  proxy_set_header Host $host;
```

```
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
  proxy_set_header X-Forwarded-Server  $host;
  proxy_set_header X-Forwarded-Host  $host;
}

location /buildbot/sse/ {
    # proxy buffering will prevent sse to work
    proxy_buffering off;
    proxy_pass http://localhost:8083/sse/;
}

# required for websocket
location /buildbot/ws {
    proxy_pass http://localhost:8083/ws/;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Origin "";

    # raise the proxy timeout for the websocket
    proxy_read_timeout 6000s;
}
```

3.5 PHPbb

phpbb is a bulletin board service written in php. In our example setup we will run it through `fpm` as a fast-cgi gateway.

Add the following to your nginx configuration:

```
location ~ \.php(/|$) {
  auth_request /auth/query_auth;

  auth_request_set $user $upstream_http_x_gsuite_user;
  fastcgi_param REMOTE_USER $user;

  # -- include /etc/nginx/snippets/fastcgi-php.conf;
  fastcgi_split_path_info ^(.+\.php)(/?.+)$;
  try_files $fastcgi_script_name =404;
  set $path_info $fastcgi_path_info;
  fastcgi_param PATH_INFO $path_info;
  fastcgi_index index.php;
  include /etc/nginx/fastcgi.conf;
  # -- end of snippets/fastcgi-php.conf

  fastcgi_pass unix:/workdir/php7.0-fpm.sock;
}
```

In order to take advantage of `oauthsub` as the authenticator, we need to install the Remote User plugin (phpbb forum page).

Download the zip file, and extract it to `phpBB/ext/cheshirekow/remoteuseauth`. Once installed, go to the administrator control panel and activate it (see the github README for screenshots).

3.6 Jenkins

Documentation coming soon!

3.7 Sonatype Nexus

Documentation coming soon!

3.8 Discourse

Documentation coming soon

CHAPTER 4

Systemd Unit

For linux servers using systemd, you can add `/etc/systemd/system/oauthsub.service`, an example which is given below assuming we want the service to run as user `ubuntu` and the configuration file is in `/etc/oauthsub.py`.

```
[Unit]
Description=oauthsub service
After=nginx.service

[Service]
Type=simple
ExecStart=/usr/local/bin/oauthsub --config /etc/oauthsub.py
User=ubuntu
Restart=on-abort

[Install]
WantedBy=multi-user.target
```

Changelog

5.1 v0.2 series

5.1.1 v0.2.1

- Almost no code changes other than minor formatting
- Added a bunch of new documentation including how to register an app with many common providers, and how to configure many common services to work behind oauthsub.

5.1.2 v0.2.0

- ported to from oauth2client (deprecated) to oauthlib
- slight refactoring into utils/appliation
- refactored application logic into a more flask-familiar layout

5.2 v0.1 series

5.2.1 v0.1.3

- python3 compatability
- add bypass option for debugging in a local environment
- cleanup package organization a bit
- add github provider support
- allow custom jinja template
- use gevent or twisted for production mode

5.2.2 v0.1.2

- Fix setup.py pointing to wrong main module, wrong keywords, missing package data
- Add Manifest.in
- Fix wrong config variable in main()

5.2.3 v0.1.1

- Fix setup.py description string

5.2.4 v0.1.0

Initial public commit

- Authenticates with google, authorizes anyone who has an email address that is part of a configurable list of domains.
- Only works with google as an identity provider
- Configuration through python config file, or command line arguments
- Includes example nginx and oauthsub configuration files
- Module directory can be zipped into an executable zipfile and distributed as a single file.

TODO

- Use the base url configuration parameter in the layout template, rather than hardcoding *auth/*.

- Decide whether or not to support custom redirects. Instead of serving up jinja renderings we could redirect to user-configured webpages. We can provide messages or additional data in cookies by adding the *Set-Cookie* header to the response before sending it out. The user defined pages can then do whatever they want with the cookie information.

oauthsub package

7.1 Module contents

This lightweight web service performs authentication. All requests that reach this service should be proxied through nginx.

See: https://developers.google.com/api-client-library/python/auth/web-app

class oauthsub.auth_service.**Application**(*app_config*)

    Bases: flask.app.Flask

    Main application context. Exists as a class to keep things local... even though flask is all about the global state.

    render_message(*message*, **args*, ***kwargs*)

    route(*rule*, ***options*)

        A decorator that is used to register a view function for a given URL rule. This does the same thing as add_url_rule() but is intended for decorator usage:

```
@app.route('/')
def index():
    return 'Hello World'
```

        For more information refer to url-route-registrations.

        Parameters

                • **rule** – the URL rule as string

                • **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint

                • **options** – the options to be forwarded to the underlying Rule object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (GET, POST etc.). By default a rule just listens for GET (and implicitly HEAD).

Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

session_get(*key*, *default=None*)
  Return the value of the session variable *key*, using the prefix-qualifed name for *key*

session_set(*key*, *value*)
  Set the value of the session variable *key*, using the prefix-qualifed name for *key*

oauthsub.auth_service.**callback**()
  Handle oauth bounce-back.

oauthsub.auth_service.**forbidden**()
  The page served when a user isn't authorized. We'll just set the return path if it's available and then kick them through oauth2.

oauthsub.auth_service.**get_session**()
  Return the user's session as a json object. Can be used to retrieve user identity within other frontend services, or for debugging.

oauthsub.auth_service.**login**()
  The login page. Start of the oauth dance. Construct a flow, get redirect, bounce the user.

oauthsub.auth_service.**logout**()
  Delete the user's session, effectively logging them out.

oauthsub.auth_service.**query_auth**()
  This is the main endpoint used by nginx to check authorization. If this is an nginx request the X-Original-URI will be passed as an http header.

oauthsub.auth_service.**strip_settings**(*settings_dict*)
  Return a copy of the settings dictionary including only the kwargs expected by OAuth2Session

class oauthsub.configuration.**Configuration**(*rooturl=None*, *flask_debug=False*, *flask_privkey=None*, *response_header=None*, *allowed_domains=None*, *host=None*, *port=None*, *logdir=None*, *flaskopt=None*, *route_prefix=None*, *session_key_prefix=None*, *bypass_key=None*, *user_lookup=None*, *client_secrets=None*, *custom_template=None*, *enable_forbidden=True*, *server=None*, ***kwargs*)

  Bases: object

  Simple configuration object. Holds named members for different configuration options. Can be serialized to a dictionary which would be a valid kwargs for the constructor.

  classmethod get_fields()
    Return a list of field names in constructor order.

  serialize()
    Return a dictionary describing the configuration.

oauthsub.configuration.**default_user_lookup**(*_*, *parsed_content*)
  Default username resolution just returns the email address reported by the provider.

oauthsub.configuration.**get_default**(*obj*, *default*)
  If obj is not *None* then return it. Otherwise return default.

class oauthsub.util.**ZipfileLoader**(*zipfile_path*, *directory*)
  Bases: jinja2.loaders.BaseLoader

  Jinja template loader capable of loading templates from a zipfile

get_source(*environment*, *template*)

> Get the template source, filename and reload helper for a template. It's passed the environment and template name and has to return a tuple in the form `(source, filename, uptodate)` or raise a *TemplateNotFound* error if it can't locate the template.
>
> The source part of the returned tuple must be the source of the template as unicode string or a ASCII bytestring. The filename should be the name of the file on the filesystem if it was loaded from there, otherwise *None*. The filename is used by python for the tracebacks if no loader extension is used.
>
> The last item in the tuple is the *uptodate* function. If auto reloading is enabled it's always called to check if the template changed. No arguments are passed so the function must store the old state somewhere (for example in a closure). If it returns *False* the template will be reloaded.

oauthsub.util.**get_zipfile_path**(*modparent*)

> If our module is loaded from a zipfile (e.g. a wheel or egg) then return the pair (zipfile_path, module_relpath) where zipfile_path is the path to the zipfile and module_relpath is the relative path within that zipfile.

Purpose

The goal of `oauthsub` is to enable simple and secure Single Sign On by deferring authentication to an `oauth2` provider (like google, github, microsoft, etc).

`oauthsub` does not provide facilities for access control. The program is very simple and if you wanted to implement authentication *and* access control, feel free to use it as a starting point. It was created, however, to provide authentication for existing services that already do their own access control.

Details

`oauthsub` implements client authentication subrequest handling for reverse proxies, and provides `oauth2` redirect endpoints for doing the whole `oauth2` dance. It can provide authentication services for:

- NGINX (via http_auth_request)
- Apache (via mod_perl and Authen::Simple::HTTP, backup link)
- HA-Proxy (via a lua extension, backup link)

The design is basically this:

- For each request, the reverse proxy makes a subrequest to `oauthsub` with the original requested URI
- `oauthsub` uses a session cookie to keep track of authenticated users. If the user's session has a valid authentication token, it returns HTTP status 200. Otherwise it returns HTTP status 401.
- If the user is not authenticated, the reverse proxy redirects them to the `oauthsub` login page, where they can start the dance with an `oauth2` provider. You can choose to enable multiple providers if you'd like.
- The `oauth2` provider bounces the user back to the `oauthsub` callback page where the authentication dance is completed and the users credentials are stored. `oauthsub` sets a session cookie and redirects the user back to the original URL they were trying to access.
- This time when they access the URL the subrequest handler will return status 200.

Oauthsub will also pass the authenticated username back to the reverse-proxy through a response header. This can be forwarded to the proxied service as a Remote User Token for access control.

Application Specifics

`oauthsub` is a flask application with the following routes:

- `/auth/login`: start of oauth dance
- `/auth/callback`: oauth redirect handler
- `/auth/logout`: clears user session
- `/auth/query_auth`: subrequest handler
- `/auth/forbidden`: optional redirect target for 401's

The `/auth/` route prefix can be changed via configuration.

`oauthsub` uses the flask session interface. You can configure the session backend however you like (see configuration options). If you share the session key between `oauthsub` and another flask application behind the same nginx instance then you can access the `oauthsub` session variables directly (including the `oauth` token object).

10.1 Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index