txcas Documentation

Release 0.1

Carl Waldbieser

August 17, 2015

Contents

Contents:

CHAPTER 1

Overview

The Central Authentication Service (CAS) is a protocol that allows a single web site to act as the authentication broker for service providers. Twisted is an asyncronous networking library for the Python programming language. Since prefixing project names with "twisted" is somewhat long-ish, Twisted Python projects tend toward using the "tx" prefix. So "txcas" is an implementation of a CAS server using the Twisted Python library.

1.1 Features

- Implements the CAS Protocol v3.0 required sections (1-3).
- Easy to start/stop service that listens and responds to incoming requests. No external web server or web application container required.
- Open source Python code making heavy use of the Twisted networking library.
- Flexible plugin architecture, allowing customization of major architectural components.
- Plugins for Authentication (file, unix, LDAP, client x509), User Realms (basic, LDAP), Ticket Stores (in-memory, CouchDB), Service Managers (JSON), and View Providers (Jinja2 templates).
- Simple configuration.
- Runs on a Raspberry Pi!

1.2 Why Another CAS Server?

The Apereo Foundation already maintains the reference CAS server implementation. It is robust, well tested, reliable, flexible software that has a vibrant community behind it. So why another server implementation?

Ultimately, the reason this project exists is that I unapologetically love programming in Python! It has been said the Python "fits your brain", and in my case, I most certainly agree. I am also a big fan of the Twisted networking library and asynchronous I/O.

I recognize many of the benefits of the Java programming language and its associated tool chain, but it is not my software environment of choice. I found a basic CAS server written in Python on GitHub. I forked it, and started this project.

1.2.1 Goals

My goals for this project are as follows:

- Produce a working, production quality CAS server that implements all the required features of the CAS protocol.
- Provide a flexible and customizable plugin architecture. Don't try to include every option in the core server.
- Keep the code base simple to learn and understand.
- Keep the administration of the service simple to use.

CHAPTER 2

Installation

1. *Clone the source from GitHub*
2. *Configure the Python environment*
3. *Create configuration files*
4. *Start the service*

2.1 Clone the source from GitHub

Use the standard **`git clone`** command:

```
$ git clone 'https://github.com/cwaldbieser/txcas.git'
```

2.2 Configure the Python environment

If you are new to Python, this will probably be the most difficult step. txcas is tested and run on Python v2.7. Older versions (e.g. v2.6) *may* work, but are not recommended.

You can download Python from the official web site. If you are running some flavor of Linux or BSD, your distribution's package manager may provide a pre-packaged Python. The official documentation has a helpful Setup and Usage section.

Note: Attention Windows users! In addition to the Python installer available from the official web site, there are some alternative bundles. The ActivePython installer is a great choice for Python on Windows!

2.2.1 Fufilling Dependencies

Warning: Fufilling dependencies tends to be where the real pain points in any software installation are felt. I apologize in advance. The good news is that you probably only have to do this once to set up a development environment. If you set up production environments from source, make sure you take good notes if this step isn't a smooth ride.

The `requirements.txt` file lists all the *Python* dependencies for txcas. Some Python modules may require dependencies on external system libraries which may vary depending on your platform. Installing all the dependencies manually is not a fun process.

While there is no silver bullet, a lot of work has been done to make satisfying dependencies a bit more civilized. Your package manager may provide python modules that you can **`yum install`** or **`apt-get install`**.

I recommend installing dependencies in a Python virtual environment. This keeps all your dependencies isolated from your system Python and any other Python environments you have. There is a handy guide to virtual environments.

Once I have a virtual environment created and activated, I use pip to install the requirements listed in `requirements.txt`.

```
$ pip install -r ./requirements.txt
```

Ideally, you can sit back and relax while the packages are downloaded from the Python Package Index (PyPi) and installed as if by magic. In practice, sometimes there are unmet dependencies external to Python that pop up. You may not have the traditional build tools for your platform installed. This will cause issues if one of the dependencies needs to build a C-extension, for example.

Missing external libraries is another common issue. Sometimes it will be necessary to install the *devel* version of a library using your package manager so the header files are available to compile against.

2.2.2 txcas on Raspberry Pi

Since you made it this far, here is an interesting tidbit. Using the above technique, I was able to install txcas on a Raspberry Pi! Using the Raspbian image I installed the following system packages usinig **`apt-get install`**:

- python-dev
- libffi-dev
- python-virtualenv
- virtualenvwrapper
- vim
- git
- htop

The first 2 were the only actual dependencies I needed to install. The *python-virtualenv* and *virtualenvwrapper* packages are just for working with Python virtual environments (see above). **`vim`** is my editor of choice when working on a Pi, **`git`** is needed to clone the txcas source, and **`htop`** is just fun to watch once txcas is up and running!

2.2.3 xubuntu 14.04 Dependencies for txcas

Here are the OS packages that needed to be installed on a new xubuntu 14.04 install to build txcas.

- build-essential
- libffi-dev
- libssl-dev
- libxml2-dev
- libxslt1-dev

2.3 Create configuration files

In the project directory, copy `cas.cfg.example` to `cas.cfg`. Edit the file and change the settings to suit your needs. Copy `cas.tac.example` to `cas.tac`. Edit the file to configure the endpoint (host, port, SSL options) on which the service will run.:

```
$ cd txcas
$ cp cas.cfg.example txcas.cfg
$ vim txcas.cfg
$ cp cas.tac.example cas.tac
$ vim cas.tac
```

Note: The `cas.tac` file is a Twisted Application Configuration (TAC) file. It is essentially a Python file used for configuring a Twisted Application. As such, it needs to conform to Python syntax. The `cas.tac` file has deliberately been kept very simple so configuration is not confusing for users who don't have a lot of familiarity with Python. Python enthusiasts should feel free to experiment with adding settings to this file. See Using the Twisted Application Framework for more information.

You may need to make additional configuration changes depending on the plugins you enable. For example, if you use the JSON service registry plugin, you will need to create a service registry file. `serviceRegistry.json.example` is included in the project root as a starting point.

2.4 Start the service

The service is started and stopped with the **`twistd`** program included with the Twisted networking library. This program is used run a Twisted Application. The simplest invocation of this command is to provide the necessary application configuration in a *TAC file*, which is a regular Python code file.

The **`twistd`** command can also to be used to configure services from the command line. In this case, the CAS service can be run as a **`twistd`** sub-command, and options specified on the command line will override options specified in configuration files.

2.4.1 Running the Service as a Twisted Application

Start the service by invoking the **`twistd`** command:

```
$ twistd -n -y cas.tac
```

The above command runs the application in the foreground. If you want to run the service as a daemon (background service), omit the `-n` option.

2.4.2 Running the Service as a twistd Subcommand

You can run the service using the *cas* subcommand to **`twistd`**. Running the service this way allows you to specify options on the command line or inspect the online help.:

```
$ twistd -n cas
```

Again, the `-n` option runs the service in the foreground. To run it as a daemon process, omit that option. If you specify the `--help` option after the *cas* subcommand, the program will output a list of options.

CHAPTER 3

Demonstration

The program **`sample.py`** included in the txcas project root can spin up a CAS service and 4 simple service providers to demonstrate various aspects of the CAS protocol. Once you have successfully installed the txcas software, you can run the demonstration with the following command:

```
$ python ./sample.py
```

You should see log entries that indicate the ports on which the services are listening. The ports are:

- 9800: The CAS service.
- 9801: Service 1. A basic service that will be used as the middle of a proxy chain.
- 9802: Service 2. A more advanced service that can obtain a *PGT* and act as a proxy.
- 9803: Service 3. A basic service that requires primary credentials and does not participate in *SSO*.
- 9804: Service 4. A basic service.

The demonstration will run without any configuration files. By default, the following plugins will be selected:

- Credential checker: In-memory database with user 'foo' and password 'password'.
- User realm: A demonstration realm that produces made-up attributes.
- Ticket store: An in-memory ticket store.

The demonstration also customizes the CAS views to some extent, but does not use a view provider or service manager.

3.1 Take The CAS Tour

Point a browser to service 1 at http://127.0.0.1:9801/ . You will be redirected to the CAS server to log in. Use 'foo' and 'password' as the credentials and you will be redirected back to the service. You will see you are now logged in as 'foo'.

The log being printed to the console will have printed out the /proxyValidate XML response, including some (fictitious) attributes that were added to the avatar by the demonstration user realm.

If you point your browser to service 2 at http://127.0.0.1:9802/, your SSO session provided by the CAS ticket granting cookie (TGC) will have transparently allowed you to log into the second service without having to re-enter crdentials.

Service 2 will also allow you to proxy service 1, which will in turn proxy service 4. The result returned will show the complete proxy chain.

Service 3 requires you to use primary credentials to log in.

3.2 Experimenting With the Demonstration

The demonstration program honors any plugin and option settings made in the main txcas configuration file. You can try out plugins and options with the demo services. If you run **`sample.py`** with the `--no-cas` command line option, the services will be started *without* the CAS service. You can run the CAS service in another console and observe how the program interact. The `--cas-base-url` option lets you specify the base CAS service URL. This is useful if you want to run the CAS service on a different host and/or port.

CHAPTER 4

Configuration

The txcas service is configured primarilly via a single configuration file. The service looks for this file at the following locations:

- `/etc/cas/cas.cfg`
- `$HOME/.casrc`
- `$PWD/cas.cfg`

The configuration options will be loaded, in order, from each of the locations. Options that appear in multiple locations will be overwritten by the values that occur later in the search order, so system-wide options will be overridden by user-specific options, which will be overridden by options specified in the current woking folder.

This configuration is in a simple INI format. Options are key-value pairs that occur one per line. Keys are separated from values by an equal sign (=). Options are grouped into sections, which are denoted by a symbol enclosed by square brackets ([]).

Sections are:

- *CAS*: This section contains general options for the service.
- *Plugins*: This section conatins options used to enable various plugins.
- *Sections Specific to Plugins*: Some plugins have unique or shared sections used for configuration.

4.1 CAS

This section is used for configuring basic CAS behavior. Options are:

- `lt_lifespan`: The length of time, in seconds, before a login ticket expires. Default 300
- `st_lifespan`: The length of time, in seconds, before a service ticket expires. Default 10
- `pt_lifespan`: The length of time, in seconds, before a proxy ticket expires. Default 10
- `pgt_lifespan`: The length of time, in seconds, before a proxy granting ticket expires. Default 600
- `tgt_lifespan`: The length of time, in seconds, before a ticket granting ticket expires. Default 86400
- `validate_pgturl`: Validate a *pgtUrl* callback certificate, as per the CAS protocol. Default is 1 (True).
- `ticket_size`: The ticket size in characters. Default 128
- `static_dir`: If this option is set to a folder, the cas service will serve static content out of this folder to the */static* resource. By default, no static content is served.

4.2 Plugins

This section is used to enable the plugins used for various parts of the service. The plugin options supported are:

- `cred_checker`: The tag used to determine the mechanism that will be used for authenticating the credentials presented to the service. If this option is not specified, the service defaults to using a file-based user database named `./cas_users.passwd`. Entries are assumed to be in *user:password* format, one entry per line.
- `realm`: The tag used to determine the plugin that will create an avatar that will be exposed to a service, mainly via attribute release. A realm receives an avatar ID that will have already been autheticated via a cred_checker. If this option is not specified, the service defaults to using a basic realm that does not include any attributes.
- `ticket_store`: The tag used to determine the plugin that will be used to manage tickets that CAS uses. If this option is not specified, the service defaults to using the in-memory ticket store.
- `service_manager`: The tag used to determine the plugin that will be used to determine whether a service is allowed to authenticate with this CAS service. A service manager also determines if the service participates in *SSO*. Extra information provided in the registry is also made available to the view_provider plugin. If a service manager plugin is not specified, CAS will run in *open* mode, and any service will be allowed to authenticate with this CAS service.
- `view_provider`: The tag used to determine the plugin that will be used to provide customized views of CAS pages. If not specified, the service will provide its own functional but lackluster views.

4.3 Sections Specific to Plugins

Some configuration sections are specific to certain plugins. Some plugins may also reference shared sections. For example, the *json_service_manager* plugin can be configured to use a particular service registry file via the section *JSONServiceManager*. The *ldap_simple_bind* cred_checker plugin and the *ldap_realm* realm plugin both reference the shared *LDAP* section to obtain LDAP-specific options.

CHAPTER 5

Endpoints

A Twisted server endpoint is the end of the connection on which a service listens for incoming requests. For simple testing and development, the configured endpoint may be a simple TCP socket. In a production setting, an SSL endpoint would be more appropriate.

Server endpoints can be described using a simple string format. Additionally, txcas provides the *tls:* endpoint which extends the standard *ssl:* endpoint with several additional options.

5.1 TLS Endpoint Options

- `sslmethod` : This option is present in the *ssl:* endpoint and allows you to set the SSL method (e.g. *TLSv1_METHOD*). The *tls:* endpoint allows you to specify multiple methods joined with '+'. E.g. *TLSv1_1_METHOD+TLSv1_2_METHOD*. Other OpenSSL options may be specified. For a complete list, see the PyOpenSSL documentation.
- `authorities` : A path to a file that contains one or more trusted CA certificates in PEM format used to verify client certificates. If this option is not specified, client certificates are not verified.
- `revokedFile` : A path to a file that contains glob patterns, one per line. Blank lines and lines starting with "#" are ignored. The files referenced by each pattern should contain one or more revoked client certificates in PEM format. These certificates are no longer trusted by the service, and the SSL/TLS handshake will fail if a client presents one to the service. The file is read once when the service is started. If the file modification time is updated, all the patterns will be re-processed. (The *NIX **`touch`** command can cause the file to be re-processed even if no pattern has been changed).

Note: By default, a TLS endpoint will negotiate one of of TLSv1.1, or TLSv1.2.

CHAPTER 6

Authentication

Authentication in txcas is implemented using a plugin system built into the core Twisted library known as Twisted Cred. This system is actually composed of 3 distinct parts: a credential checker, a portal, and a user realm.

The credential checker is the component that accepts primary credentials and authenticates them. If successful, it returns an avatarID that the user realm will use to produce an *avatar*.

Currently, txcas supports accepting simple username/password credentials as well as a client certificte checker (trust-based authentication).

A number of credential checkers are available in Twisted Cred that support the username/password credential type. txcas also includes support for the ldap_simple_bind credential checker via the ldaptor library.

6.1 Authentication Phases

It is possible for authentication to happen in one of two distinct phases. The phase that occurs first is the *credential requestor* (or *cred_requestor*) phase. This happens when the user browser makes an HTTP GET request to the txcas service */login* endpoint. At this point, it is possible to attempt trust-based authentication *before* the login page is rendered. If successful, a user will never see the login page. Username/password based authentication is not available in this phase as the user has not ye had a chance to enter credentials.

The second phase is the *credential acceptor* (or *cred_acceptor*) phase. This phase happens when the user's browser makes an HTTP POST to the txcas service */login* endpoint with a username and password. **Both** trust-based authentication and username/password authentication may take place in this phase. If a trust-based credential checker is configured to authenticate during this phase, it will attempt authentication first. If successful, the resulting *avatar ID* is compared to the username that was submitted. If they do not match, authentication will fail. Otherwise, username/password authentication will take place. Only if **both** forms of authentication succeed will authentication be successful.

6.1.1 Typical Models For Trust-Based Authentication

Due to the fact that trust-based authentication can be configure to occur in either authentication phase, the user experience can vary.

In the **Trust-Only** model, trust based authentication is the only option. Only a trust-based credential checker is configured. There is no username/password credential checker. The trust-based checker should be configured to occur in the cred_requestor phase. A user will be authenticated if her browser has a valid certificate. If not, an error page would be presented. The user would never see a login page.

The **Trust-or-Login** model, a trust-based checker is enabled in the cred_requestor phase. A username/password checker is also enabled (this can *only* occur in the cred_acceptor phase). If the user's browser has a valid certificate,

the user is authenticated transparently as in the "Trust-Only" model. If not, the user will be presented with the txcas login view and be able to authenticate with a username/password.

In the **Trust-and-Login** (a kind of *two factor authentication*), the trust checker is enabled in the cred_acceptor phase and a username/password checker is also enabled. In this case, authentication will *only* succeed if the user's browser has a valid certificate *and* she enters a valid username/password *and* the username she supplies matches the *avatar ID* extracted from the certificate.

Configuration

An authentication method is selected via the `cred_checker` option in the *PLUGINS* section of the main configuration file. Valid options are:

- `memory`: An in-memory password database suitable for demonstrations and development. Do **not** use for production!
- `file`: A file containing *username:password* entries, one per line. This option should be followed by a colon and the path to the file. E.g. *file:/etc/cas/cas_users.passwd*.
- `unix`: Attempts to authenticate against a user on the local UNIX-like system.
- `ldap_simple_bind`: Attempts a simple BIND against an LDAP server. The LDAP options can be configured by appending a colon to this option and providing colon-separated key=value pairs *or* by configuring options in the LDAP section of the main config file (the latter method is preferred).

  The initial connection to the server may be unecrypted or encrypted depending on the client endpoint specification used (**tcp** vs. **ssl**). Although an initial SSL connection is supported by many directories (the so-called **ldaps** scheme) this type of connection is not included in the LDAP protocol RFCs. Instead, the LDAP protocol supports STARTTLS, which establishes a TLS connection *after* the initial connection is made.

  Note: StartTLS should *not* be used in conjunction with an SSL/TLS endpoint. Because it establishes a TLS connection in response to a protocol request, the initial connection should occur on an unencrypted TCP endpoint.

  A 2-stage BIND is used when checking credentials. In stage 1, an service DN and password are used to BIND in order to search for the target entry. If the target entry is located, this authenticator attempts to BIND using the password supplied at the CAS login.

  The LDAP options are:

  - `endpointstr`: A Twisted endpoint specification describing the client connection to the LDAP service.
  - `basedn`
  - `binddn`
  - `bindpw`
  - `query_template`: Defaults to *(uid=%(username)s)*. The query template is a filter that will be used by the LDAP service to identify the entry that it will attempt to BIND as using the supplied password. The *%(username)s* part of the filter will be substituted with the provided username in order to produce the final filter. The username will be escaped according to LDAP filter rules. The default template attempts to locate an entry where the *uid* attribute matches the provided username. If no matching entry is located, or if multiple matching entries are located, authentication will fail.
  - `start_tls`: (Default 0). 1=use StartTLS. 0=don't use StartTLS.
  - `start_tls_hostname`: If the expected hostname of the directory service is not specified, the StartTLS connection will be encrypted, but not verified. This will leave the connection vulnerable to man-in-the-middle (MITM) style attacks.

  - `start_tls_cacert`: Typically, this option is not required as the LDAP client will use CA certificates based on an OS-specific trust mechanism (platform trust). However, if the directory you connect to uses an internal CA certificate, you may specifically indicate a file in PEM format that contains the CA certificate to trust when using StartTLS..

- `client_cert`: This form of authentication is trust-based and happens during a SSL handshake. In order for this checker to succeed, the txcas service must run on a TLS endpoint. At least one CA certificate that the server will trust for client certificates must be specified via the `authorities` option for the endpoint. Multiple certificates may be concatenated in the authorities file, but the user experience may be degraded. A browser will typically ask the user to select the certificate that ought to be presented to the server if multiple valid options are available.

  The options for this plugin are:

  - `subject_part`: The part of the subject to extract, e.g. "CN", or "emailAddress".
  - `transform`: A comma-separated list of 'upper', 'lower', 'strip_domain'. One or more transforms are applied to the extracted subject part..
  - `auth_when`: The authentication phase when this checker is active. Valid options are 'cred_requestor' (default) and 'cred_acceptor'.

If you have added additional plugins to your `$TXCAS/twisted/plugins` folder, additional option values may be available. The plugin documentation should cover these. You can also list the available plugins with the following command:

```
$ twistd -n cas --help-auth
```

CHAPTER 7

User Realms

In txcas, a user realm is a component that translates an authenticated avatarID into an *avatar* that implements the *ICASUSer* interface. The CAS service will use this object when determining the username and attributes that should be sent to a service provider during a */serviceValidate* or */proxyValidate* request. A user realm object will implement the Twisted Cred *IRealm* interface.

The separation of authentication and *avatar* generation allows avatars to be populated with attributes that are not neccessarilly available via the authentication provider. For example, txcas could be configured to authenticate against a file-based password database, but the avatar could be populated with attributes retrieved from a web-based service or an LDAP directory.

A realm is enabled by setting the `realm` option of the *PLUGINS* section in the main configuration file. The realm options included in txcas are:

- `demo_realm`: A realm created for demonstration purposes. The avatar is constructed directly from the avatarID (username) provided and populated with phony attributes.
- `basic_realm`: This realm is very basic and suitable for situations where attribute release is not needed. In this case, CAS acts as a pure authenticator, and service providers must base access control decisions entirely on the avatar username.
- `ldap_realm`: This realm constructs the avatar by BINDing to a LDAP directory and retrieving a set of possible attributes.

  The LDAP options can be configured by appending a colon to this option and providing colon-separated key=value pairs *or* by configuring options in the LDAP section of the main config file (the latter method is preferred).

  The initial connection to the server may be unecrypted or encrypted depending on the client endpoint specification used (**tcp** vs. **ssl**). Although an initial SSL connection is supported by many directories (the so-called **ldaps** scheme) this type of connection is not included in the LDAP protocol RFCs. Instead, the LDAP protocol supports STARTTLS, which establishes a TLS connection *after* the initial connection is made.

  Note: StartTLS should *not* be used in conjunction with an SSL/TLS endpoint. Because it establishes a TLS connection in response to a protocol request, the initial connection should occur on an unencrypted TCP endpoint.

  The LDAP options are:

  - `endpointstr`: A Twisted endpoint specification describing the client connection to the LDAP service.
  - `basedn`
  - `binddn`
  - `bindpw`

- `query_template`: Defaults to *(uid=%(username)s)*. The query template is a filter that will be used by the LDAP service to identify the entry that it will attempt to BIND as using the supplied password. The *%(username)s* part of the filter will be substituted with the provided username in order to produce the final filter. The username will be escaped according to LDAP filter rules. The default template attempts to locate an entry where the *uid* attribute matches the provided username. If no matching entry is located, or if multiple matching entries are located, avatar generation will fail. txcas will report this as an authentication failure to end users, though the logs should be helpful in determining the reason.
- `attribs`: A comma separated list of attributes that the realm should attempt to populate during avatar generation.
- `aliases`: A comma separated list of aliases that is the same length as the `attribs` option. Each attribute fetched will be mapped to the alias name indicated.
- `service_based_attribs`: 1 (True) or 0 (False). Defaults to False. If this option is selected *and* a service manager plugin is used, the service entry for the current service will be used to look up a list of attributes or a mapping of attributes-to-aliases. Whether a list or a mapping, the data should be located under the **attributes** key of the service registry entry. If that key is not present for a particular entry the `attribs` and `aliases` options above will be used to compute the attributes to add to the realm.
- `start_tls`: (Default 0). 1=use StartTLS. 0=don't use StartTLS.
- `start_tls_hostname`: If the expected hostname of the directory service is not specified, the StartTLS connection will be encrypted, but not verified. This will leave the connection vulnerable to man-in-the-middle (MITM) style attacks.
- `start_tls_cacert`: Typically, this option is not required as the LDAP client will use CA certificates based on an OS-specific trust mechanism (platform trust). However, if the directory you connect to uses an internal CA certificate, you may specifically indicate a file in PEM format that contains the CA certificate to trust when using StartTLS..

If you have added additional plugins to your `$TXCAS/twisted/plugins` folder, additional option values may be available. The plugin documentation should cover these. You can also list the available plugins with the following command:

```
$ twistd -n cas --help-realms
```

CHAPTER 8

Ticket Stores

Ticket stores in txcas are plugins used to keep track of the various tickets used by the CAS protocol. Ticket stores generate tickets on request. A ticket store must track how long a ticket is valid and expire it appropriately. A ticket store is also responsible for validating tickets, and making Single Log Out (*SLO*) callbacks to services.

A ticket_store is enabled by setting the `ticket_store` option of the *PLUGINS* section in the main configuration file. The ticket_store options included in txcas are:

- `memory_ticket_store`: This ticket store manages tickets entirely in the memory allocated to the txcas process. It has the advantage of being quite fast when it comes to ticket creation, modification, or expiration. There is no network latency. However, this ticket store is limited in that it is not persistant. If the process is stopped and restarted, all tickets that were previously in the ticket store are lost. Also, for situations where CAS servers span multiple nodes, this type of ticket store cannot be shared across process or server boundries.
- `couchdb_ticket_store`: This ticket store manages tickets in an external CouchDB database. This ticket store may have network latency issues associated with it that are not present in an in-memory ticket store. However, tickets stored "in the couch" are persistant. Because the ticket storage is external, tickets can be shared across multiple nodes. Also, CouchDB's master-master replication capabilites make this storage worthy of consideration for high availability scenarios. Projects like CouchDB Lounge or Big Couch are certainly worth a look if scalability is a concern.

  Because CouchDB is written to be operated completely with a RESTful API, no special database drivers are required. It is also a good fit with the Twisted asynchronous I/O model.

  The CouchDB options can be configured by appending a colon to this option and providing colon-separated key=value pairs *or* by configuring options in the *CouchDB* section of the main config file (the latter method is preferred).

  The CouchDB options are:

  - `host`: The database server hostname or IP address.
  - `port`: The port that CouchDB listens on.
  - `db`: The name of the database (e.g. "cas_tickets").
  - `user`: The username to connect to the database as.
  - `passwd`: The password to use when connecting to the database.
  - `https`: 1 (True) or 0 (False). When connecting to the database, use HTTPS.
  - `verify_cert`: 1 (True) or 0 (False). When connecting to the database, verify its X509 cert. It is useful to set this option to False during development if using a self-signed cert.

8.1 Options Common to All Ticket Stores

All ticket stores must support specific options:

- `lt_lifespan`: The time in seconds before a Login Ticket expires.
- `st_lifespan`: The time in seconds before a Service Ticket expires.
- `pt_lifespan`: The time in seconds before a Proxy Ticket expires.
- `pgt_lifespan`: The time in seconds before a Proxy Granting Ticket expires.
- `tgt_lifespan`: The time in seconds before a Ticket Granting Ticket expires.
- `ticket_size`: The size of a ticket (in characters) generated by the ticket store.

Note: Ticket lifespan countdowns for multi-use tickets (*PGT* s and *TGT* s) may be reset if a ticket is used. Some tickets have their lifespans connected to their parent tickets as per the CAS protocol and should **not** outlive their parent tickets.

8.2 Interaction With Service Managers

If a service manager is enabled in the txcas service, the ticket store will use it to determine if the CAS server will authenticate for a particular service.

CHAPTER 9

Service Managers

Service managers are plugin components within txcas that determine whether CAS will validate tickets for a particular service provider. Service managers also determine if services that CAS will validate will participate in CAS *SSO* sessions. If a service manager determines that a given service will **not** participate in *SSO*, then primary credentials will *always* be requested via the CAS login whenever authentication is requested for that service.

Service managers are free to provide additional information about services. This information may be consumed by a view provider plugin, if one is enabled.

Service managers are enabled by setting the `service_manager` option in the *PLUGINS* section of the main configuration file. Valid settings for this option include:

- json_service_manager: This service manager stores information in JSON format in a file accessible to the txcas service. The file is read, parsed, and represented in memory. If the file is changed, the service manager will detect the change and reload the file contents.

  The options for this plugin can be configured by appending a colon to the option name and providing colon-separated key=value pairs *or* by configuring options in the JSONServiceManager section of the main config file (the latter method is preferred).

  The JSONServiceManager options are:

  - `path`: The path to the service registry JSON file.

  The format of the the service registry is a list of entries, where each entry is a mapping of key-value pairs. The following keys have special meanings to the service manager:

  - *name*: The name of the service. Used mainly for identification during logging.
  - *scheme*: One of *http*, *https*, or *.
  - *netloc*: A value composed of a host or domain pattern, and optionally followed by a colon and a port number. If the port number is omitted, it is inferred by the actual scheme of the service (443 for https, 80 for http). A host/domain pattern is in dotted notation. Each component of the name may be replaced by an asterisk (*) indicating that component is a wildcard match. If the first component is a double asterisk, that means that *any* hostname that ends with the same pattern will match.
  - *path*: The path part of the service URI.
  - *child_paths*: *true* or *false*. whether to include child paths of the path component as matches. False indicates an exact path match is required.
  - *required_params*: A mapping of required parameters in key:list-of-values format or *null*. If the required parameters and values are not present, the service will not match.
  - *SSO*: *true* or *false*. If false, CAS will authenticate the service, but it will request primary credentials each time. The service will not participate in *SSO*.

CHAPTER 10

View Providers

View provider plugins render the user facing web pages in txcas. These include:

- The login page
- The successful login to SSO page
- The logout page
- The invalid service page
- The error page
- The resource not found page

If a service manager is enabled, a reference to it is given to a view provider so that a service entry is made available to the *login page* and *invalid service* views.

A view provider does not have to provide every view. If it does not provide a particular view, the default txcas view will be presented.

A view provider is enabled by setting the `view_provider` option in the *PLUGINS* section of the main configuration file. Valid options include:

- *jinja2_view_provider*: This view provider renders HTML pages from Jinja2 templates. The *request* object is made available to all templates. The following names are made available to each view:
  - *login page*
    - *login_ticket*: A login ticket that must be POSTed when presenting credentials.
    - *service*: The service requesting authentication. May be an empty string, indicating the user is trying to log into a CAS *SSO* session without logging into a service.
    - *service_entry*: The complete service entry from the service manager.
    - *failed*: True / False, indicates if previously submitted credentials failed.
    - *request*
  - *successful login*
    - *avatar*: The avatar provided by the user realm.
    - *request*
  - *logout*
    - *request*
  - *invalid service*

    - *service*: The service requesting authentication.
    - *service_entry*: The complete service entry from the service manager.
    - *request*
  - *error*
    - *err*: The failure object.
    - *request*
  - *not found*
    - *request*

The plugin options can be configured by appending a colon to this option and providing colon-separated key=value pairs *or* by configuring options in the *Jinja2ViewProvider* section of the main config file (the latter method is preferred).

The *Jinja2ViewProvider* options are:

- `template_folder`: The path to the folder that will contain the templates. The templates should be named:
  - `login.jinja2`
  - `login_success.jinja2`
  - `logout.jinja2`
  - `invalid_service.jinja2`
  - `error_5xx.jinja2`
  - `not_found.jinja2`

CHAPTER 11

Plugin Troubleshooting

If you install a plugin but don't see it listed as a valid option, you can try running the **`./plugin_test.py`** script from the main project folder. This script is a simple diagnostic that lists all available plugins of the types relevant to txcas. Pay special attention to any error output produced, as it may indicate some kind of problem with the plugin installation.

CHAPTER 12

Development

The txcas software makes heavy use of the Twisted Plugin System. The core software implements the web interactions and delegates various operations to plugins.

12.1 Basic File Layout and Script Requirements

Plugin intergration code should be located in `$PROJECT/twisted/plugins` in a Python script file. The script should assign global variables to instances of a class or classes that implement the factory interface for the plugin you are developing. For example, a file called `/somewhere/on/your/PYTHONPATH/myspecialticketstore.py` might look something like:

```
from txcas.interface import ITicketStore, ITicketStoreFactory
from twisted.plugin import IPlugin
from zope.interface import implements

class WickedCoolTicketStoreFactory(object):
    """
    A factory for creating wicked-cool ticket stores!
    """
    implements(IPlugin, ITicketStoreFactory)

    tag = "wicked_cool_ticket_store"
    opt_help = "I am detailed help printed on the command line."
    opt_usage = "I am the brief help printed on the command line."

    def generateTicketStore(self, argstring=""):
        """
        This method returns an object that implements ITicketStore.
        It is configured via the string passed to this function, but it can
        also pull settings out of the txcas configuration file.  The
        settings passed in should be given preference to those in the
        config file.
        """
        # Implementation is something you would need to code.
        # ...
        return a_shiny_new_ticket_store
```

The file `$TXCAS_ROOT/twisted/plugins/myspecialticketstoreplugin.py` should contain something like:

```
from myspecialticketstore import WicketCoolTicketStoreFactory
import txcas.settings

aplugin = WickedCoolTicketStoreFactory()
```

Note: In the above example, the script that actually implements the `WickedCoolTicketStoreFactory` does not need to reside in the txcas project folder. It can be located anywhere on your `PYTHONPATH`.

The code that instantiates the plugin factory *should* reside in the `$TXCAS_ROOT/twisted/plugins` folder.

The reason that factories are used is that many plugins tend to need some kind of configuration. Factories can be created with no configuration, and they can accept command line arguments that can be used in the configuration process.

If you look at the source code in `txcas/interface.py`, you will see that for each plugin type, there is an interface for a factory and an interface for the plugin the factory produces.

For more information on writing Twisted plugins, see Writing a twistd Plugin

12.2 Kinds of Plugins

All unqualified interface references below are understood to belong to the *txcas.interface* module.

Credential checkers and **user realms** are are components of Twisted Cred, Twisted's pluggable authentication system. Credential checkers authenticate credentials presented. User realms create *avatar* s for authenticated users. Currently, txcas supports credential checkers that consume credentials that implement the twisted.cred.credentials.IUsernamePassword interface.

Credential checker factories should implement the twisted.cred.strcred.ICheckerFactory interface. Credential checkers should implement the twisted.cred.checkers.ICredentialsChecker interface. User realm factories should implement the *IRealmFactory* interface. User realms should implement the twisted.cred.portal.IRealm interface. Avatars produced by a realm should implement the *ICASUser* interface.

Some credential checkers are able to operate based on information that is present in either of the 'cred_requestor' or the 'cred_acceptor' authentication phases. These plugins should implement the *ICASAuthWhen* interface to communicate to the server the phase in which the plugin should be active.

Ticket stores manage the tickets used by CAS. They track ticket lifetimes, validate them, and expire them. Ticket stores may need to work with *service managers* to determine if a ticket ought to be created for a service provider, or if a service provider participates in *SSO*.

Ticket store factories should implement the *ITicketStoreFactory* interface. Ticket stores should implement *ITicketStore*.

Service managers are used to decide whether a service provider is allowed to authenticate with a particular txcas instance, and whether or not a service provider will participate in *SSO*. Without a service manager, txcas runs "open", meaning that **any** service provider may authenticate with it.

Service manager factories should implement *IServiceManagerFactory*. Service managers should implement *IServiceManager*.

Service managers may also provide additional service entry meta-data that other plugins can use. This meta-data may be used to customize views or activate decision making logic in other components (e.g. the attributes included in a realm could be tailored to specific services). If a plugin wants to receive a reference to the service manager, it should implement the *IServiceManagerAcceptor* interface.

View providers are used to customize the web pages presented by the txcas service. This kind of customization makes it possible to present a specific theme or appearance that meshes with an organizational web site.

View provider factories should implement *IViewProviderFactory*. View providers should implement *IViewProvider*. A view provider's `provideView()` method should return a callable if it provides a particular view or *None* if it does not.

12.3 Unit Tests

txcas comes with its own unit tests. To run the tests:

```
$ trial txcas/test/test_server.py
```

You should see a number of test cases with statuses for each test: **SKIPPED**, **FAIL** or **OK**. Tests that are skipped typically require some kind of middleware to be running that is difficult to emulate for the test. An example would be the CouchDB ticket store. These tests tend to be slow and require configuration information to be passed to the test script. To enable these tests, copy the file `txcas/test/tests.cfg.example` to `txcas/test/tests.cfg`. Edit the *Tests* section to enable the optional tests. Provide any required settings for the middleware in the appropriate section and re-run the tests.

When developing your own plugins, it is recommended you develop your own unit tests. For more information on unit testing with Twisted, see the Trial documentation and its associated howto.

CHAPTER 13

Glossary

AVATAR A representation of an authenticated user. In txcas, an avatar must implement the interface ICASUSer. The avatar will have a username and possibly one or more attributes associated with it.

AVATAR ID An identifier that can be used to uniquely represent an *avatar*. A username submitted from a login form could be an example of an avatar ID.

PGT Proxy Granting Ticket. A ticket obtained by a service provider that allows it to request proxt tickets from CAS. The proxy tickets can later be used to request services from other service providers that participate in the CAS session.

SLO Single Log-out. When a user is logged out of a CAS *SSO* session, all CAS clients that authenticated via the session are notified of the session termination.

See https://github.com/Jasig/cas/blob/master/cas-server-protocol/3.0/cas_protocol_3_0.md#233-single-logout for details.

SSO Single Sign-On. The ability to login once to a service authentication broker and not have to present primary credentials to log into same or different participating services, often for a specific period of time.

TAC FILE A Twisted Application Configuration file. A regular Python file used to configure a Twisted Application. Endpoint settings (interface, port, SSL settings) are commonly configured in this type of file.

TGT Ticket Granting Ticket. A ticket issued when a CAS session is started by providing primary credentials. The TGT is then used to request service tickets that a service provider can validate with CAS to prove that the ticket presenter has been authenticated by CAS.

TWO FACTOR AUTHENTICATION Authentication based on two independent authentication factors. Factors may include something known to a user, something a user has, or something a user is. See http://en.wikipedia.org/wiki/Multi-factor_authentication for more information.

CHAPTER 14

Indices and tables

- genindex
- search