Galaxy Auth Documentation

- Configure Auth
  - Universe_wsgi.ini
    - auth_config_file
    - auth_debug
  - Auth_conf.xml
    - Processing order
    - TYPE
      - ActiveDirectory
      - AlwaysReject
      - LocalDB
    - FILTER
    - Common options
- Writing custom authentication providers
  - Python code

Configure Auth

To add (or use) alternate Authentication modules within galaxy you need to configure it to your setup in auth_conf.xml file.

Universe_wsgi.ini

There are two optional changes to the master config file.

auth_config_file

The name of the config file that the auth module uses. Defaults to config/auth_conf.xml. This is relative to the GALAXYROOT directory.

auth_debug

Enables a bunch of extra logging output to help identify what is going wrong with the auth modules. This will cause the config options to be displayed in the log which will include system passwords for at least the ActiveDirectory module so should not be enabled on a production system.

Auth_conf.xml

This file controls how the auth module functions. See the auth_conf.xml.sample file in GALAXYROOT/config directory for some examples of how this might look.

```
<?xml version="1.0"?>
<auth>
	<authenticator>
		<type>TYPE</type>
		<filter>FILTER</filter>
		<options>
			<auto-register>BOOL2</auto-register>
			<allow-register>BOOL3</allow-register>
			<allow-password-change>BOOL2</allow-password-change>
			PROVIDER_SPECIFIC_OPTIONS
		</options>
	</authenticator>
	...
</auth>
```

Figure: syntax of auth_conf.xml file

Processing order

The authentication providers are executed in the order that they (authenticator tag) are presented in the file. The ordering is important as some providers can cause an immediate halt to authentication whereas others allow auth to continue down the chain.

TYPE

There are a number of builtin authentication providers. Sites can also implement their own providers and install them in the GALAXYHOME/lib/galaxy/auth/providers package.

The following Authenticators are built into Galaxy:

ActiveDirectory

A provider that uses ldap to search and bind to a Microsoft Active Directory. If you cannot bind using the email address provided by the user then it has the option to search the Active Directory for their username and then bind with it. The presence of the search-filter option enables the search functionality.

Table: ActiveDirectory provider specific options

Option	Description
server	The connection URL used by both the search and bind functions. e.g. ldap://ADDOMAIN.company.com
search-filter ^	The LDAP search filter used to find the required user details to bind to AD. e.g. (&(objectClass=user)(mail={username})).
search-base ^	The basename of the AD. e.g. dc=ADDOMAIN,dc=company,dc=com
search-user ^	The username to use to bind to the AD in order to search it. e.g. galaxy@ADDOMAIN.company.com
search-password ^	The password for above username
search-fields ^	A comma-separated list of AD fields to retrieve in search. e.g. sAMAccountName
bind-user ^#	The username to perform to check (by binding with it). e.g. {sAMAccountName}@ADDOMAIN.company.com
bind-password ^#	The password for above account e.g. {password}
continue-on-failure	Should later authentication provides be tried if this one fails (any error including wrong password, or configuration error). In most cases this should be False

auto-register-username ^#	The format for the galaxy username when auto-registering the user. e.g. {sAMAccountName}

^ allows substitutions of {username} and {password} (which are the values typed by user)
# allows substitutions of {ADFIELD} where ADFIELD is a field specified in search-fields option.

```
<authenticator>
    <type>activedirectory</type>
    <filter>'{username}'.endswith('@company.com')</filter>
    <options>
        <allow-register>No</allow-register>
        <auto-register>Yes</auto-register>
        <server>ldap://ADDOMAIN.company.com</server>
        <search-filter>(&amp;(objectClass=user)(mail={username}))</search-filter>
        <search-base>dc=ADDOMAIN,dc=company,dc=com</search-base>
        <search-user>galaxy@ADDOMAIN.company.com</search-user>
        <search-password>SOMESECRET</search-password>
        <search-fields>sAMAccountName</search-fields>
        <bind-user>{sAMAccountName}@ADDOMAIN.company.com</bind-user>
        <bind-password>{password}</bind-password>
        <continue-on-failure>False</continue-on-failure>
        <auto-register-username>{sAMAccountName}</auto-register-username>
    </options>
</authenticator>
```

Figure: example configuration for ActiveDirectory provider using username lookup

```
<authenticator>
    <type>activedirectory</type>
    <filter>'{username}'.endswith('@ADDOMAIN.company.com')</filter>
    <options>
        <allow-register>No</allow-register>
        <auto-register>Yes</auto-register>
        <server>ldap://ADDOMAIN.company.com</server>
        <bind-user>{username}</bind-user>
        <bind-password>{password}</bind-password>
        <continue-on-failure>False</continue-on-failure>
        <auto-register-username>{sAMAccountName}</auto-register-username>
    </options>
</authenticator>
```

Figure: example configuration for ActiveDirectory provider using direct bind method

AlwaysReject

A simple provider that rejects all email addresses that match its filter. This is useful for blacklisting certain addresses.

LocalDB

Uses the local Galaxy DB to check their username/password combination. This is used in the default configuration of the auth module.

FILTER

An optional filter which decides whether or not to attempt this provider. This is a python string that is evaluated (within a restricted python environment) and if it equals True then this auth provider is executed. This can be useful for speeding up authentication by not trying providers that will fail for a given username. The field will accept {username} and {password} substitutions.

Common options

These options are common to all providers

Option	Description
auto-register	*True* = This provider should automatically register users when they first login. *False** = denies access and requests they register.
allow-register	*True** = Yes, allow registration (ignores password check) *False* = No, not allowed *Challenge* = allow registration if their username/password checks out
allow-password-change	*True* = users can change their password (only supported by LocalDB module) *False** = users cannot change password (gives them an error message saying so)

Writing custom authentication providers

This sections goes through the details required to create your own auth provider. To add a new provider you need to create the following two files and place them in the GALAXYHOME/lib/galaxy/auth/providers package (don't add them to git; they should be ignored anyway)

Python code

To implement your own provider you simply create a new python class that extends the 'AuthProvider' class. The class needs to contain (a minimum of) one attribute and implement two methods which are described in the example below. The module needs to contain one variable.

The attribute (plugin_type) needs to contain a string value that matches the module name (e.g. 'goodguysonly') without the '.py'

The first method (authenticate) takes a username (email)/ password and is used during registration when the Authenticator is set to "Challenge" mode (allow-register option).

The second method (authenticateUser) performs a similar task but takes a user object instead of a username. This method is used during regular authentication to check their username/password match.

Finally the module needs to contain the proper __all__ variable so that they auth module can find the module implementation. This variable needs to be set to a list of strings and one of these strings matching the name of the providers class (e.g. 'GoodGuysOnly')

The naming convention used here is the filename should be all lowercase with the same name as the class inside it. e.g. goodguysonly.py contains GoodGuysOnly class

```
import re, string
from ..providers import AuthProvider

import logging
log = logging.getLogger(__name__)

class GoodGuysOnly(AuthProvider):
    '''A simple authenticator to demo how to build your own'''

    plugin_type = 'goodguysonly'

    def authenticate(self, username, password, options):
        '''
        See abstract method documentation
        '''
        log.debug("Username: %s" % username)
        log.debug("Options: %s" % options)

        return (username == "goodguy@galaxy.org" && password == "secret",
                self._stripUsername(username))

    def authenticateUser(self, user, password, options):
        '''
        See abstract method documentation
        '''
```

```
        log.debug("User: %s, GoodGuysOnly: %s" %
            (user.email, user.email == "goodguy@galaxy.org" && password == "secret"))

        return (user.email == "goodguy@galaxy.org" && password == "secret",
                self._stripUsername(username))

    def _stripUsername(self, username):
        pattern = re.compile('[\W\-]+')
        return pattern.sub(username, string.printable)


__all__ = ['GoodGuysOnly']
```

Figure: example implementation of a custom authentication provider (goodguysonly.py)