Authentication in OpenStack

Jorge L Williams <jorge.williams@rackspace.com>
Khaled Hussein <khaled.hussein@rackspace.com>
Ziad N Sawalha <ziad.sawalha@rackspace.com>

Abstract

The purpose of this blueprint is to define a standard for authentication in OpenStack that enables services to support multiple authentication protocols in a pluggable manner. By providing support for authentication via pluggable authentication components, this standard allows OpenStack services to be integrated easily into existing deployment environments. It also provides a path by which to implement support for emerging authentication standards such as OpenID. The standard is not an authentication system onto itself, but rather a protocol by which authentication systems may be integrated with OpenStack services.

Table of Contents

Rationale and Goals 1
Specification Overview 1
    Deployment Strategies 2
    Exchanging User Information 3
    Reverse Proxy Authentication 3
    Handling Direct Client Connections 4
    Using Multiple Authentication Components 5
    The Default Component 5
Questions and Answers 8
References 9

Rationale and Goals

Currently, OpenStack does not support a unified authentication mechanism for its storage and compute services. This complicates the deployment of these services in a single environment and prevents OpenStack from easily integrating with existing authentication and identity management systems. To address this issue, we propose a standard for authentication that allows support for multiple authentication protocols via pluggable *authentication components*.

In this blueprint, we define the responsibilities of authentication components. We describe how these interact with underlying OpenStack services and how existing services can be modified to take advantage of pluggable authentication. Finally, we illustrate an implementation of a simple authentication component. The goal is to allow OpenStack services to be integrated easily into existing deployment environments and to provide a path by which to implement support for emerging authentication standards such as OpenID.

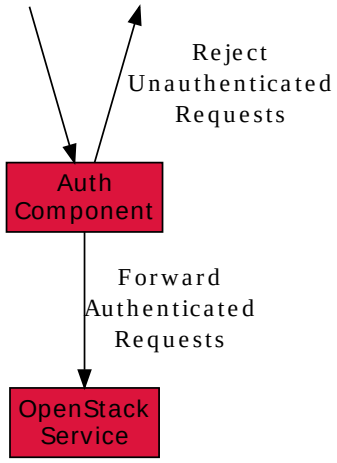
Specification Overview

Authentication is the process of determining that users are who they say they are. An *authentication protocol* may obtain user information via a variety of back-end services such as identity management systems, LDAP directories, relational databases, and flat files; current authentication protocols include

HTTP Basic Auth, Digest Access, public key, token, etc. An *authentication component* is a software module that implements an authentication protocol.

At a high level, an authentication component is simply a reverse proxy that intercepts HTTP calls from clients. If the authentication component verifies the user's identity, the authentication component extends the call with information about the current user and forwards the call to the OpenStack service. Otherwise, the message is rejected before it gets to the service. This is illustrated in Figure 1. Note that, in this blueprint, we define interactions between the authentication component and the OpenStack service. Interactions between the client and the authentication component are defined only for exceptional cases. For example, we define the message that should be returned when the OpenStack service is down. Other interactions, however, are defined by the underlying authentication protocol and the OpenStack service and are outside of the scope of this blueprint.

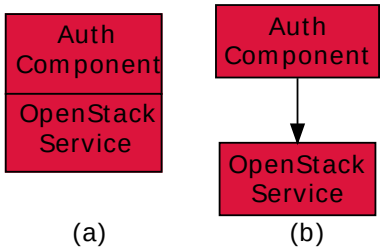
Figure 1. An Authentication Component



Deployment Strategies

An authentication component may be integrated directly into the service implementation, or it may be deployed separately as an HTTP reverse proxy. This is illustrated in Figure 2 showing both approaches to authetication, labeled Option (a) and Option (b).

Figure 2. Authentication Component Deployments Options



Option (a) The authentication component is integrated into the service implementation;
Option (b) The authentication component is deployed in a separate node.

In Option (a), the component is integrated into the service implementation. In this case, communication between the authentication component and the service can be efficiently implemented via a method call.

In Option (b), the component is deployed separately and communication between the service and the component involves an HTTP request. In both cases, unauthenticated requests are filtered before they reach the service.

Each approach offers some benefits. Option (a) offers low latency and ease of initial implementation, making it possibly most appropriate as a starting point for simple configurations. Option (b) offers several kep advantages that may be of a particular value in complex and dynamic configurations. It offers the ability to scale horizontally in cases where authentication is computationally expensive, such as when verifying digital signatures. Option (b) also allows authentication components to be written in different programming languages. Finally, Option (b) allows multiple authentication components to be deployed in front of the same service.

OpenStack services **MUST** support both embedded (Option (a)) and external (Option (b)) deployment strategies. Individual authentication components **MAY** support either strategy or they **MAY** support both strategies. In order to support option (a), authentication components written in the Python programming language **MUST** be written as middleware components in accordance to the Web Server Gateway Interface (WSGI) standard [1]. Additionally, services **MUST** support the ability to swap between different embedded or external authentication components via configuration options.

Exchanging User Information

If a request is successfully authenticated, the authentication component **MUST** extend the request by adding an `X-Authorization` header. The header **MUST** be formatted as illustrated in Example 1.

Example 1. An X-Authorization Header

```
X-Authorization: Proxy JoeUser
```

Here, `Proxy` denotes that the authentication occurred via a proxy (in this case authentication component) and *`JoeUser`* is the name of the user who issued the request..

Note

We are considering using an `Authorization` header rather than an `X-Authorization` thereby following normal HTTP semantics. There are some cases, however, where multiple `Authorization` headers need to be transmitted in a single request. We want to assure ourselves that this will not break common clients before we recommend the approach.

Authentication components **MAY** extend the request with additional information. For example, an authentication system may add additional headers or modify the target URI to pass authentication information to the back-end service. Additionally, an authentication component **MAY** strip sensitive information from the request — a plain text password, for example; That said, an authentication component **SHOULD** pass the majority of the request unmodified.

Reverse Proxy Authentication

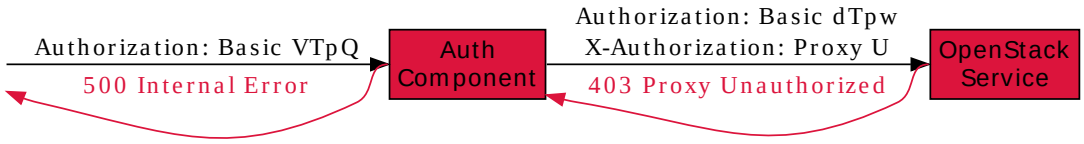
It is important for an OpenStack service to verify that it is receiving requests from a trusted authentication component. This is particularly important in cases where the authentication component and the OpenStack service are deployed separately. In order to trust incoming requests, the OpenStack service should therefore authenticate the authentication component. To avoid confusion, we call this *reverse proxy authentication* since in this case the authentication component is acting as an HTTP reverse proxy.

Any HTTP-based authentication scheme may be used for reverse proxy authentication; however, all OpenStack services and all authentication components **MUST** support HTTP Basic Authentication as defined in RFC 2617 [2].

Whether or not reverse proxy authentication is required is strictly a deployment concern. For example, an operations team may opt to utilize firewall rules instead of an authentication protocol to verify the integrity of incoming request. Because of this, both OpenStack services and authentication components **MUST** also allow for unauthenticated communication.

In cases where reverse proxy authentication is used, the authorization component may receive an HTTP 401 authentication error or an HTTP 403 authorization error. The authentication component **MUST NOT** return these errors to the client application. Instead, the component **MUST** return a 500 internal error. This is illustrated in Figure 3. The component **SHOULD** format the error in a manner that does not break the service contract defined by the OpenStack service.

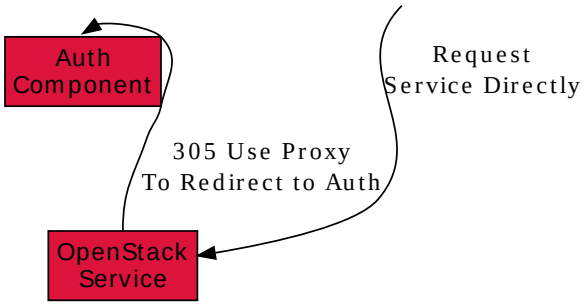
Figure 3. Reverse Proxy Authentication



Handling Direct Client Connections

Requests from the authentication component to an OpenStack service **MUST** contain an `X-Authorization` header. If the header is missing, and reverse proxy authentication fails or is switched off, the OpenStack service **MAY** assume that the request is coming directly from a client application. In this case, the OpenStack service **MUST** redirect the request to the authentication component by issuing an HTTP 305 User Proxy redirect. This is illustrated in Figure 4. Note that the redirect response **MUST** include a `Location` header specifying the authentication component's URL as shown in Example 2.

Figure 4. Auth Component Redirect



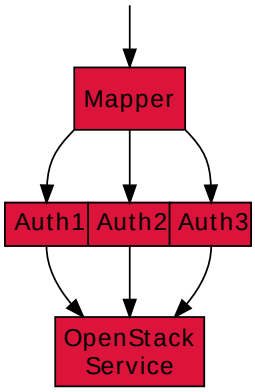
Example 2. Auth Component Redirect Response

```
HTTP/1.1 305 Use Proxy
Date: Thu, 28 Oct 2010 07:41:16 GMT
Location: http://sample.auth.openstack.com/path/to/resource
```

Using Multiple Authentication Components

There are some use cases when a service provider might want to consider using multiple authentication components for different purposes. For instance, a service provider may have one authentication scheme to authenticate the users of the service and another one to authenticate the administrators or operations personnel that maintain the service. For such scenarios, we propose using a Mapper as illustrated in Figure 5.

Figure 5. Multiple Authentication Components



At a high level, a Mapper is a simple reverse proxy that intercepts HTTP calls from clients and routes the request to the appropriate authentication component. A Mapper can make the routing decisions based on a number of routing rules that map a resource to a specific authentication component. For example, a request URI may determine whether a call should be authenticated via one authentication component or another.

Note that neither the authentication component nor the OpenStack service need be aware of the mapper. Any external authentication component can be used along side others. Mappers may provide a means by which to offer support for anonymous or guest access to a subset of service resources. Finally, note also that a mapper may be implemented via a traditional reverse proxy server such as Pound or Zeus.

The Default Component

Individual services **MUST** be distributed with a simple integrated authentication component by default. This default component **MUST** offer support for HTTP Basic Access Authentication where usernames and passwords are stored in a flat configuration file under `/etc/openstack/users.ini`. The format of the file is illustrated in Example 3. The file associates a username with a SHA-1 digest of the user's password. The SHA-1 digest is used to avoid exposing the password as plain text. If the file is missing the default component **MUST** reject all requests. We provide a reference implementation of a default authentication component, and describe it in some detail below.

Example 3. Example users.ini

```
[users]

user:5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
user2:2aa60a8ff7fcd473d321e0146afd9e26df395147
user3:1119cfd37ee247357e034a08d844eea25f6fd20f
.
.
.
```

There are a number of benefits to standardizing on a simple auth component for all OpenStack services:

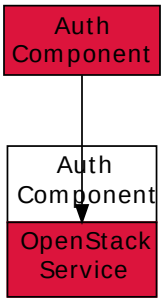
1. Providing an easy to configure embedded authentication component by default, lowers barriers to the deployment of individual services. This is especially important to developers who may want deploy OpenStack services on their own machines.

2. Since there is no direct dependency to an external authentication system, OpenStack services can be deployed individually — without the need to stand up and configure additional services.

3. Having a standard authentication component that all services share promotes a separation of concerns. That is, as a community we are explicitly stating that services should not develop their own authentication mechanisms. Additional authentication components may be developed, of course, but these components should not be intimately coupled to any one particular service.

Note that services should maintain support for option (b), even as we require that they ship with a simple embedded authentication component by default. One way to achieve this is to provide a method that allows the disabling of the default authentication component via configuration. This is illustrated in Figure 6. Here, requests are sent directly to the OpenStack service when the default authentication component is disabled.

Figure 6. Disabled Embedded Component



Implementation Details

The reference implementation is composed of a number of files:

`authcomp.py` The script contains an implementation of a default authentication component. The `authcomp.py` script is capable of running as

	a stand alone reverse proxy to illustrate a distributed deployment strategy.
`service.py`	Provides an example of a very simple OpenStack service that supports both, direct integration and distributed deployment of an authentication component as defined by this blueprint.
`embedded.py`	A script that illustrates how the authentication component and example service defined above can be integrated into a single application.
`helper.py`	This script contains utility code shared by multiple scripts in the reference implementation.

For the sake of brevity we will examine only the relevant portions of the `authcomp.py` and `service.py` scripts.

Example 4. Authenticating a request in `authcomp.py`

```
def __call__(self, environ, start_response):
 if not environ.get("HTTP_AUTHORIZATION"):                    ❶
  # The user needs to be authenticated. We reject the request and
  # return 401 before the Service (MyApp) receives the request.
  return respond401(environ, start_response)                  ❷
 else:
  # Let's authenticate the user against the users.ini file.
  import base64
  auth_header = environ['HTTP_AUTHORIZATION']
  auth_type, encoded_creds = auth_header.split(None, 1)
  username, password  = base64.b64decode(encoded_creds).split(':', 1)
  if self.validateCreds(username, password):                  ❸
   # The Auth Component has to authenticate itself to the service.
   environ['HTTP_AUTHORIZATION'] = "Basic dTpw"               ❹
   # The Auth Component passes the username to the Service
   environ['HTTP_X_AUTHORIZATION'] = "Proxy %s" % username    ❺
   return self.app(environ, start_response)                   ❻
  else:
   return respond401(environ, start_response)                 ❼
```

❶❷ Every request that flows through the authentication component is checked for authentication credentials. The request is rejected if the credentials are missing.

❸ Next, the credentials are validated against the `users.ini` file. Note that this involves taking a SHA-1 of the input password and comparing it against the digest in the `users.ini` file.

❹❺ If the credentials validate, the authentication component extends the request by adding it's own authorization credentials. It also adds the `X-Authorization` header which contains the username of the user who initiated the request.

❻ The authentication component then forwards the request to the OpenStack service.

❼ If the user's credentials don't validate, the request is rejected before it is sent to the OpenStack service.

Example 5. Verifying and service a request in `service.py`

```
def __call__(self, environ, start_response):
 if not environ.get("HTTP_X_AUTHORIZATION"):
  return respond305(environ, start_response, \                    ❶
   createurl(environ, self.proxy_host, self.proxy_port))

 # Now let's authenticate the Auth Component itself
 if not environ.get("HTTP_AUTHORIZATION"):
  return respond401(environ, start_response)

 import base64
 auth_header = environ['HTTP_AUTHORIZATION']
 auth_type, encoded_creds = auth_header.split(None, 1)
 # Not very secure, but you get the picture
 if str(encoded_creds) != "dTpw":
  return respond401(environ, start_response, False)               ❷
 else:
  # Here is where the service processes the request
  response_headers = [('content-type', 'text/html')]
  response_status = '200 OK'
  user_auth_header = environ["HTTP_X_AUTHORIZATION"]
  auth_type, username = user_auth_header.split(None, 1)
  response_body = "Welcome %s to OpenStack <BR>" % username       ❸
  start_response(response_status, response_headers)
  return [response_body]
```

❶ The service verifies that the `X-Authorization` is present in the request. If it isn't it redirects the client via a 305 to the authentication component.

❷ If reverse proxy authentication is enabled, the service validates the credentials sent by the authentication service. If these are missing, or don't match, the service issues a 401 error.

❸ Otherwise, the service responds to the request. In this case, the service extracts the username from the `X-Authorization` header.

Questions and Answers

1. Why not have the authentication component pass authentication failures back to the service instead of rejecting requests up front?

The content and format of an authentication failed message is determined by the authentication scheme (or protocol). In order for the service to respond appropriately it would have to be aware of the authentication schemes it's participating in — this defeats the concept of pluggable authentication components.

2. Why require support for deploying authentication components in separate nodes?

The deployment strategy is very flexible. It allows for authentication components to be horizontally scalable. It allows for components to be written in different languages. Finally, it allows different authentication components to be deployed simultaneously as described above.

References

[1] Phillip J Eby. *Python Web Server Gateway Interface v1.0*. http://www.python.org/dev/peps/pep-0333/.

[2] J Franks. P Hallam-Baker. J Hostetler. S Lawrence. P Leach. A Luotonen. L Stewart. *HTTP Authentication: Basic and Digest Access Authentication*. http://tools.ietf.org/html/rfc2617.