Develop and test Web authentication with containers

Jan Pazdziora, Red Hat

1. Authentication in Web applications

Web applications that aim to be used in large enterprises and organizations need to be able to use user identities from various external identity sources within those organizations, like FreeIPA/IdM or Active Directory domains, or in case of federated setups even make use of user identities provided by completely independent entities — customer or partner organizations. Users are then authenticated via authentication methods and protocols like Generic Security Service Application Program Interface (GSS-API) / Kerberos or Security Assertion Markup Language (SAML). External identity sources can also hold and make available additional pieces of information for authorization of access to those Web applications, for instance user group membership which can map to application-specific user roles.

Many Web applications start small. Initially they might be targeted just at limited number of users so they tend to include some in-application user and user group management and handling. That is often already implemented by frameworks or development environments. For example, the Django Web framework populates its default `startproject` configuration result with `django.contrib.auth` and `django.contrib.admin`, supporting both authentication and internal user identity management features out of the box.

When external authentication is needed by first large deployment, support is quickly added for that particular protocol or mechanism, often addressing the bare minimum of features. LDAP backend support might get quickly added but without proper TLS or failover handling. Similar approach might repeat for each next protocol. Web frameworks can make this work easier but the applications developers still need to be on the lookout for new protocol requirements.

2. Offloading authentication operations

To minimize the impact that the new and evolving authentication requirements have on the development team, it is often beneficial to move the authentication operations out of the application and framework code, to a front end HTTP server, and extend the application to be able to consume the authentication and authorization results.

When Apache HTTP Server is used as the authentication front end, wealth of modules can be used for various external authentication mechanisms:

Module	**Protocol**
mod_auth_gssapi	Negotiate / GSS-API / Kerberos
mod_ssl / mod_nss	SSL client / X.509 / smart-card authentication
mod_auth_mellon	SAML
mod_auth_openidc	OpenID Connect
mod_authnz_pam	Pluggable authentication modules (PAM)

The authentication for the particular application deployment can be configured in the Apache HTTP Server to use a certain module or module combination, even if the application developer team never tried that setup themselves. All that is needed is for the application to get the authentication result and act accordingly.

Of course, some rudimentary familiarity of the external protocols and their impact especially on the HTTP traffic is useful. In case of Negotiate / Kerberos authentication, HTTP responses status 401 together with `WWW-Authenticate: Negotiate` response header drives the authentication flow. For SAML or OpenID Connect, HTTP redirects to authentication providers are used. With SSL/TLS, the authentication happens (and can fail) even before the HTTP traffic is processed, while the connection is being established. When developing and testing, some external services are needed for most protocols, and for cases like Kerberos, configuration on clients might be needed as well. That configuration might however clash with the production Kerberos and other configuration used on developers' machines.

To help developers get experience with these environments and protocols, container-based Web application authentication developer setup with isolated "multihost" setup was created.

3. Web application authentication developer setup

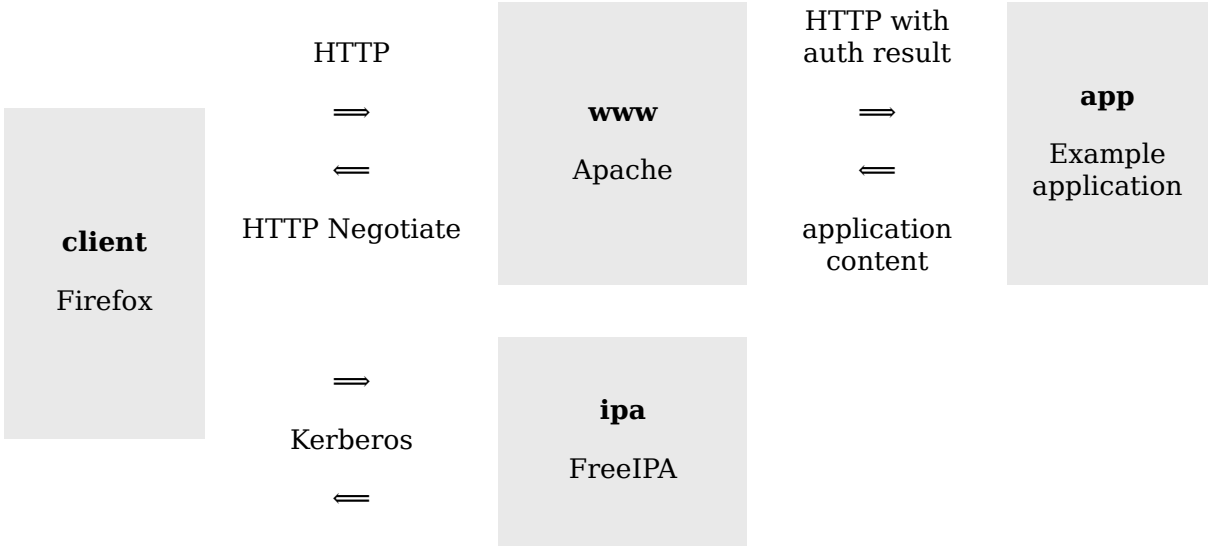
As of October 2016, the default setup consists of four containers:

Container	Purpose
ipa	FreeIPA identity management server + DNS server + Ipsilon SAML Identity Provider (IdP)
www	Apache HTTP Server authentication front-end; authentication via Apache modules
client	Firefox and Kerberos command-line tools on IPA-enrolled machine
app	Example Django application which demonstrates not just authentication behaviour but also handling of additional user attributes and use of group membership for application roles

They all run in isolated domain .example.test, managed by the integrated DNS server in the **ipa** container.

3.1. Kerberos operation

By default, Kerberos authentication via mod_auth_gssapi in HTTP authentication proxy is enabled, which leads to the following HTTP Negotiate authentication workflow:



In addition to the authentication, mod_lookup_identity with SSSD is configured in the **www** container to retrieve additional attributes about the authenticated user, like name, email address, or user group membership, and pass it to the application.

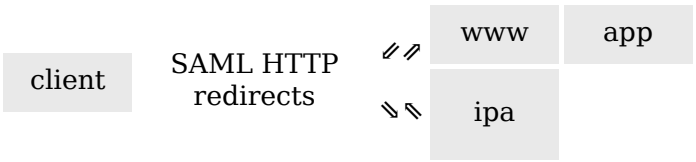
The **client** container runs SSH daemon so it is possible to start Firefox via `ssh -X`. In the same container (thus on http://localhost/ from the Firefox point of view) there is Web front-end to some Kerberos utilities: `kinit`, `klist`, `kpasswd`, `kdestroy`. That makes it easy to study the behaviour of HTTP Negotiate and Kerberos credential caches without diving into command line. That of course is also available and so is `curl`.

3.2. SAML operation

The source repository contains alternative configuration of the Apache HTTP Server authentication front-end, SAML Service Provider (SP) using mod_auth_mellon. The **www** container itself is already configured as SP for the IdP in the **ipa** container, so moving the setup from Kerberos to SAML has only two steps:

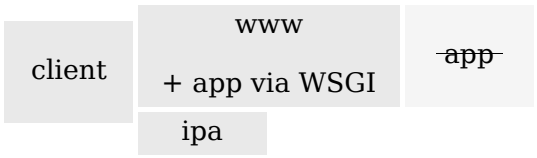
- Copy the provided Apache config file to the container data directory.
- Restart the Apache HTTP Server daemon.

Instead of HTTP Negotiate authentication and Kerberos communication, HTTP redirects for the SAML protocol will be then used:



3.3. Application on the HTTP server machine

Many Web application deployments run the application on the same machine as the HTTP end point. The developer setup supports this runtime style as well. The example application can run in the **www** container via mod_wsgi, instead of its own container via HTTP proxying:



This setup allows studying and debugging interaction of applications deployed under the Apache HTTP Server, where authentication results and additional information is passed via environment variables and similar mechanisms, instead of HTTP request headers.

As for the authentication itself, both Kerberos and SAML configurations can be used.

4. Developer setup usage

So far we have been describing the stock developer setup and its use with the example application. While that can be useful for learning about the protocol internals, developers will be more interested in ways of using the setup for their own applications that they develop. Let us look at the options. In all cases, the **app** container is no longer needed and can be disabled altogether.

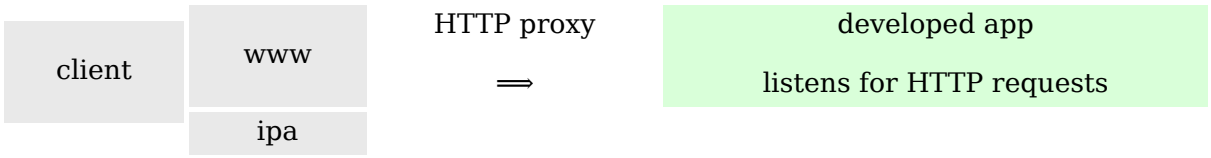
4.1. HTTP proxy

Let us assume that the developed or tested application runs at its own HTTP endpoint. By merely changing the target of the default proxy directives

```
ProxyPass / http://app.example.test/
```

```
ProxyPassReverse / http://app.example.test/
```

the authentication front-end in the **www** container can be pointed to application running on the host, in different container, or on completely different machine.

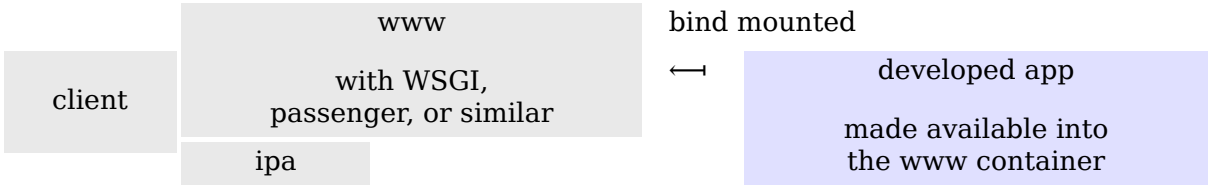


For correct operation, the configuration of the logon locations (URIs) will need to be tuned as well, to match the workflow and locations expected and supported by the application.

4.2. Application under the HTTP server

As mentioned earlier, in many deployment cases, applications will be executed by and under the HTTP server context, for instance with mod_wsgi of Apache HTTP Server. In that case, the best use of the developer setup is to run the application in the **www** container. Extending and rebuilding the setup might be needed to bring in packages of the required modules, language interpreters, or runtime environments — Ruby / passenger, Java / tomcat / AJP, etc.

The code of the developed application, potentially the working tree checkout with the source code, can then be bind-mounted to the **www** container to the location where the Apache modules will be configured to find it:

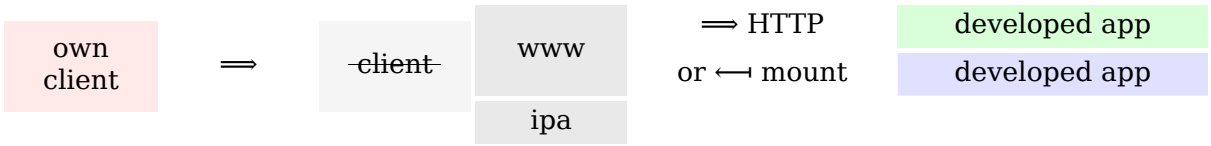


Alternatively, for example for testing, the application code can be installed to the container image during build time. Like in the previous case, the logon locations (URIs) as seen by the Apache HTTP Server and as expected by the application will need to be configured to match the application workflow.

4.3. Using own clients

The developer setup includes **client** container with Firefox and Kerberos tools installed, IPA-enrolled and part of the example.test domain. That allows for simple start, without additional configuration needed.

For development and use in continuous integration and other automated environments, other clients can of course be also used. They might need additional configuration to properly resolve hostnames of the servers in the developer setup or find the Kerberos infrastructure. For this, the integrated DNS server in the **ipa** container can be used. The **client** container automatically IPA-enrolls itself, so keytab file can be copied to those other sources.



5. Developer setup internals

As of October 2016, the containers of the setup are based on Fedora 24. Except for the **app** container, they are all systemd-based: the **ipa** container runs multiple services and both

www and **client** IPA-enroll themselves with `ipa-client-install` which needs systemd to be run.

The first execution of the setup takes a couple of minutes (depending on the performance of the machine it is run on) because the FreeIPA server needs to be configured, using `ipa-server-install`. The configuration and data is then stored in the `ipa-data/` directory, so subsequent starts are significantly faster.

The other containers also configure themselves and their configuration and data are stored in `www-data/`, `client-data/`, and `app-data/`, respectively.

The **client** container is configured to start Firefox browser via `ssh -X`, rather than bind-mounting `/tmp/.X11-unix`. This decision was made to support more flexible and versatile, albeit potentially slower, usage.

The Web application authentication developer setup is available under the Apache License, Version 2.0.

6. Building the developer setup

The developer setup is defined as a Docker Compose application. The repository contains `docker-compose.yml` which describes the individual services — containers. Those are defined in `src/Dockerfile.*` files.

The **ipa** container's `src/Dockerfile.ipa` expects that `freeipa-server` container image is available. It is possible to build the image from sources from the `docker-freeipa` repository, pull built images from Docker registry and tag as `freeipa-server`, or tweak the `FROM` line in the `Dockerfile` to name a specific image to use.

The container images for the setup are then built with single

```
$ docker-compose build
```

command.

7. Running the developer setup

Once the images are built, command

```
$ docker-compose up
```

will start the containers. The output will likely begin with lines similar to

```
Creating webauthinfra_ipa_1
Creating webauthinfra_app_1
Creating webauthinfra_www_1
Creating webauthinfra_client_1
Attaching to webauthinfra_app_1, webauthinfra_ipa_1, webauthinfra_www_1, webauthinfra_c
app_1     | + echo password32345
app_1     | + cp -p /var/www/django/project/db.sqlite3 /data/db
app_1     | + cd /var/www/django/project
app_1     | + python manage.py shell
app_1     | ++ cat /data/admin-password
app_1     | + echo 'from django.contrib.auth.models import User; User.objects.create_su
app_1     | Python 2.7.12 (default, Aug  9 2016, 15:48:18)
app_1     | [GCC 6.1.1 20160621 (Red Hat 6.1.1-3)] on linux2
app_1     | Type "help", "copyright", "credits" or "license" for more information.
app_1     | (InteractiveConsole)
app_1     |
app_1     | + grep '^SECRET_KEY' project/settings.py
```

```
app_1     | + cd /var/www/django/project
app_1     | + REMOTE_USER_VAR=HTTP_X_REMOTE_USER
app_1     | + python manage.py runserver app.example.test:80
ipa_1     | Configuring ipa.example.test ...
ipa_1     | /usr/sbin/ipa-server-configure-first
www_1     | Waiting for FreeIPA server (HTTP Server) ...
client_1  | Waiting for FreeIPA server (HTTP Server) ...
ipa_1     |
ipa_1     | The log file for this installation can be found in /var/log/ipaserver-insta
ipa_1     | ==============================================================================
ipa_1     | This program will set up the FreeIPA Server.
ipa_1     |
ipa_1     | This includes:
ipa_1     |   * Configure a stand-alone CA (dogtag) for certificate management
```

It shows that the **app** container quickly finishes its configuration, and **www** and **client** wait for **ipa** container to configure the FreeIPA server, to then allow them to IPA-enroll and finish their configuration.

After the startup finishes, FreeIPA server's admin password can be found in the `ipa-data/admin-password` file and private SSH key of user `developer` in the **client** container is stored in `client-data/id_rsa`. To start process in the **client** container, we find out its IP address with `docker inspect webauthinfra_client_1` and the SSH to it as user `developer`:

```
$ ssh -i client-data/id_rsa developer@172.18.0.5
```

By default the SSH daemon also listens on port 55022 on the host, so

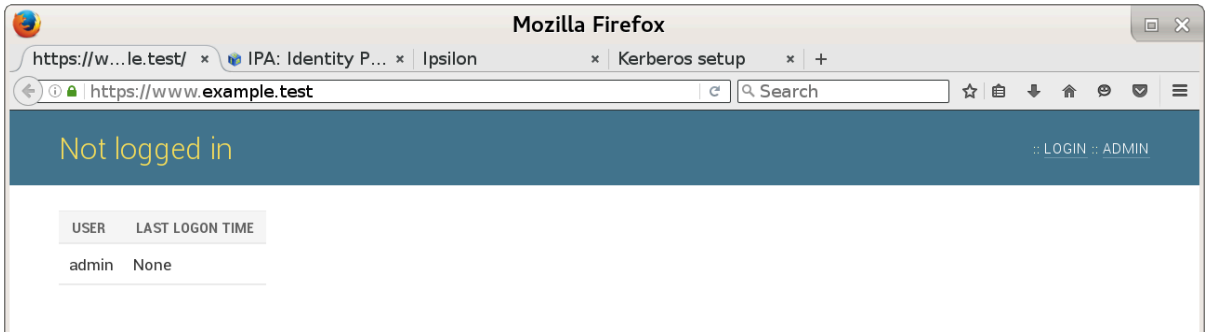
```
$ ssh -i client-data/id_rsa -p 55022 developer@localhost
```

is also possible.

To start the Firefox browser in the setup, we enable the X11 forwarding:

```
$ ssh -X -i client-data/id_rsa -p 55022 developer@localhost firefox -no-remote
```

The browser will open four tabs: the authenticated front-end to the example application, FreeIPA and Ipsilon IdP server logon pages, and a Web interface to Kerberos commands in the **client** container:



The Web application authentication developer setup is now ready in its default, Kerberos-enabled configuration.

8. Conclusion and future work

The container-based Web application developer setup provides an isolated environment which helps to study, develop, and test external authentication and authorization in Web applications.

In the future, configuration templates for more authentication methods might be added. We will also look at providing another example application written in different language / framework. Different software might be installed and configured in future versions to achieve the same functionality, for instance in the SAML IdP area.

Since the setup uses well-known realm, domain, and host names, it might be possible to run the initial FreeIPA server configuration (`ipa-server-install`) in build time. Unfortunately, due to the systemd-based nature and the build time environment, those investigations have not so far resulted in success. We will likely revisit that option in the future.

References

Web application authentication developer setup sources. https://pagure.io/webauthinfra

Developer setup presentation. https://www.adelton.com/webauthinfra/presentation/

FreeIPA container images sources. https://github.com/adelton/docker-freeipa

Automated builds of FreeIPA container images. https://hub.docker.com/r/adelton/freeipa-server/

Web App Authentication notes. https://www.freeipa.org/page/Web_App_Authentication