FreeNAS plugins (everything you ever wanted to know)

John Hixson

john@ixsystems.com
iXsystems, Inc

Abstract

When FreeNAS entered the 8.x series, many people were not happy that functionality that previously existed was no longer included. Such functionality was mainly multimedia focused and targeted at the home user. Services such as bittorrent, DLNA and iTunes media services are the primary examples. Beginning with FreeNAS 8.2.0, a plugin architecture was introduced. This architecture allows FreeNAS systems to be extended in any way that the user sees fit. The purpose of this paper is to describe the technical details of how the architecture works so that plugin authors have the knowledge to create new plugins. As a working example, the transmission bittorrent client plugin will be reviewed.

1 Introduction

FreeNAS is a very powerful open source operating system based on FreeBSD. However, once you get beyond all the great capabilities it offers, your options for extending it become limited. Your choices are using FreeBSD's built in package management system, or modifying the source code and building your own image.

Packages can be installed using FreeBSD's package management system, but care must be taken. You must be aware of what paths and files the package management system uses as well as the package itself. You have to very carefully select whats are used and where all the files go because once the system is rebooted, several key files can be overwritten or disappear.

FreeNAS creates memory disks for /var and /etc at boot time and copies the contents of /conf/base/var and /conf/base/etc to these file systems FreeBSD's package tools and ports work with files from /var/db/pkg and /var/db/ports. Also, the root file system is mounted read only. What this means is that when attempting to install a package, most files won't be allowed to be written to the system and the record in /var/db will be erased on boot. This can of course all be circumvented, but the point is that it's an involved process to get working right.

The major problem with using package management is that once you do an upgrade, everything you installed will get wiped out. Upgrades to FreeNAS only save the configuration and the volumes that are created, everything else is wiped clean.

The other option is to hack the build system to include the packages you want. This is certainly an option. The caveat with this is that you must have a FreeBSD system with development tools, an understanding of the build system and how it works, knowledge of what files to edit, and so on. This simply isn't feasible for most people. Most FreeNAS users simply aren't technical enough for this.

To address these problems and more, FreeNAS has introduced the plugin system. The plugin system is modular, self contained and allows everyday users to install programs that fit their needs onto FreeNAS from an easy to use interface. This also allows users to use their FreeNAS system as more than just a file server.

2 The plugins jail

In order to install plugins on FreeNAS, a plugins jail must first be installed, configured and running. A FreeNAS plugins jail is a standard FreeBSD jail packaged as a PBI and preconfigured with several necessary packages that allow the stock plugins to work. The plugins jail can be found in the plugins directory under the FreeNAS release directory that is being used.

To install a plugins jail, you must first upload it. This can be done from the web interface under services->plugins. You must specify where the jail will be stored temporarily when it's uploaded. The next step requires you to configure a jail path, a jail name, IP address and netmask, and a plugins archive path. The plugin jail configuration is stored in the database in the table services_pluginsjail. The following describes each column and what it is used for:

- jail_path – The file system path where the jail resides
- plugins_path – The file system path where the plugins reside
- jail_mac – MAC address for the jail interface (if configured)
- jail_ipv4address – The IPv4 address for the jail
- jail_ipv4netmask – The IPv4 netmask for the jail
- jail_name – The name of this plugins jail

Currently, Only a single IPV4 address is supported. In the future, multiple Ipv4 and Ipv6 addresses will be supported, as well as multiple plugin jails.

When the plugin jail is uploaded and configured, pbi_add is run and the jail is extracted to the jail_path + jail_name. Once this is done, the plugins jail is ready to be run. When you turn the service on, /etc/rc.d/ix-jail is invoked. This script generates the proper /etc/rc.conf lines to configure the jail with vnet and allows /etc/rc.d/jail to start the jail. Once the jail is up and running, plugins are ready to be installed.

3 Installing a plugin

FreeNAS plugins use the PC-BSD PBI9 format. plugins are installed using the web interface. Installing a plugin is very easy, navigate to Services->plugins->Install Plugin. When a plugin is installed, the PBI information is stored in the database in the table plugins_plugins which has the following columns:

- plugin_version – plugin version number
- plugin_enabled – enabled/disabled status
- plugin_ip – fastcgi server IP address
- plugin_port – fastcgi server port
- plugin_arch – i386 or amd64
- plugin_api_version – RPC API version
- plugin_name – name of the plugin
- plugin_pbi_name – PBI file name as uploaded
- plugin_path – where in the file system the plugin is installed

Once the PBI information is saved, an oauth secret and key are generated record in the services_rpctoken table. This table contains the columns:

- secret – the oauth secret
- key – the oauth key

Once the PBI and Oauth information is recorded in the database, the following steps occur:

1. The PBI is installed into the plugins jail in /usr/pbi/${plugin}-${arch}/
2. The oauth key and secret are written into /usr/pbi/${plugin}-${arch}/.oauth
3. The plugin information is written into plugins.conf which is included by nginx.conf. This tells nginx that all URL's that specify the plugin path are to be passed to the plugins fastcgi server.
4. The plugins control script is started in the jail (/usr/pbi/${plugin}-${arch}/control start). This starts the plugin fastcgi server on the IP/port combination recorded in the database.
5. The web interface will refresh. The navtree makes a request to the plugin's _s/treemenu and treemenu-icon methods. The treemenu method returns a description of how to display the plugin information in the navtree. The treemenu-icon method passes the icon for the plugin to the navtree. Once these methods are called, the plugin appears in the navtree menu under Services->plugins->${plugin} with the plugin icon. The plugin will also appear under the Services->plugins menu in the main interface.

4 How they work

When the plugin icon is clicked, django matches the plugin URL and sends the request to the plugin fastcgi server. Requesting a plugin method is of the form:

- base_url + “/plugins/” + ${plugin} + “/” + ${method}

The methods that are available are:

- edit – edit the plugin configuration
- treemenu-icon – icon to be displayed in the navtree
- _s/treemenu – what/how to display in the navtree
- _s/start – start the plugin
- _s/stop – stop the plugin
- _s/status – status of the plugin

Plugins have access to the base system via RPC calls. All RPC requests are signed with the oauth credentials given to the plugin at install time. The following RPC methods are available:

- api.version() - get the plugin API version
- plugins.plugins.get() - get a listing of installed plugins
- plugins.jail_info() - get information about the plugins jail
- plugins.is_authenticated() - test if the plugin is currently authenticated
- fs.mountpoints.get() - get a listing of available files systems
- fs.mounted.get() - get a list of mounted file systems
- fs.mount() - mount a file system into the jail
- fs.umount() - unmount a jailed file system
- fs.directory.get() - get a directory listing
- fs.file.get() - get a file
- os.arch() - get OS architecture
- api.test() - verify RPC calls are working

When an RPC request to the base system takes place, the following thins happen:

1. An RPC request is built of the form: base_url + “/plugins/json-rpc/v1”
2. The RPC request is signed with the oauth credentials
3. The RPC request is sent with the requested method
4. The method is invoked if the oauth credentials are correct and the method exists. The results are then returned back to the plugin

The fastcgi server accepts the plugin request, then dispatches accordingly. This allows anything that anything that can communicate the fastcgi protocol to be a plugin, or even to manage plugins. Because of this flexibility, plugins can be developed using any language or framework one wishes to use. All that is required for a FreeNAS plugin to work is that it implement the described methods and be packaged using the PBI9 format.

5 Making a plugin

Currently, making a plugin for FreeNAS is somewhat cumbersome. This process is expected to be streamlined in coming releases. While there are several methods to create a plugin, the one described was used to develop the 3 reference plugins included on Sourceforge.

Documentation for creating PBI files using the PBI9 format already exists, so only the FreeNAS specific portions will be covered. Creating a PBI for FreeNAS requires FreeBSD 8.x, PC-BSD 8.x, or FreeNAS 8.2.0 or higher. In all cases, pbi-manager and the ports collection must be installed. The basic procedure for creating a plugin is this:

1. Create plugin directory: myplugin
2. Create resource and scripts directories under this directory: myplugin/resources and myplugin/scripts
3. Create a PBI configuration file: myplugin/pbi.conf
4. Edit the pbi.conf file for your particular plugin Documentation for how to do this can be found at wiki.pcbsd.org “PBI Module Builder Guide”. It's relatively straight forward.
5. If there are any pre/post script needs, create the necessary scripts in the scripts directory as specified in the PBI module builder guide.
6. Invoke pbi_makeport to create the PBI

At this point, a PBI will have been created. Upload the PBI as previously described and it will be installed into the plugins jail. It will not be functional from within the web interface, but it is ready to be worked on from within the jail. This process can be repeated as the plugin is refined and developed.

Next, a control script must be created. The name of the script must be “control” and it must be located in the plugin directory (/usr/pbi/${plugin}-${arch}/. The control script takes 3 arguments, an action verb, an IP address, and a port. The purpose of the script is to start a fastcgi server on the specified IP address and port. The verbs that must be implemented are start, stop and status. The start verb starts the fastcgi server on the IP/port combination. The stop verb stops the server. The status verb exits with 0 if the server is running otherwise it exits with 1. This script is called from the main system when the plugin is enabled or disabled.

Once the control script is completed, the interface portion of the plugin can be worked on. The job of the interface is to export the methods needed by FreeNAS to integrate with the web interface as described in section 4. The start and stop methods must provide a means by which to start and stop the binary the plugin is in control of. This also includes any modifications to /etc/rc.conf if necessary. The treemenu method simply dumps out JSON. The treemenu-icon outputs the plugin icon. The workhorse of a FreeNAS plugin is the edit method. This is the method that presents the interface for configuring the plugin This generally entails saving and restoring state and generating and modifying configuration files.

6 An example - Transmission

When FreeNAS released 8.2.0, three reference plugins were also released. They were provided for two reasons: to provide the missing functionality that previously existed in FreeNAS 7.x, and to document and demonstrate how future plugins could be made.

One of the available plugins is Transmission. Transmission is a very popular bittorrent client. It's implementation is pretty simple and straight forward so it will be used for the example. Since Transmission is built into the build system, the build system configuration will be covered as well. Reviewing the build system process for making a plugin is recommended anyhow for plugin authors so they have a better understanding of how everything works. Here is an overview of the directory layout and key files for the transmission plugin:

${freenas}/nanobsd/plugins/

This is top level directory for all FreeNAS plugins. All plugin files are located in this directory. The build system will be aware of a plugin once it is placed in this directory. For Transmission, the following file is created:

${pluginroot}/transmission

This is the nanobsd configuration file for Transmission. It sets up the nanobsd environment for the Transmission build and provides function(s) for doing so. Since nanobsd is being used for the plugin build, a bit of trickery is done here. All of the nanobsd functions are overridden with stub calls except the last_orders() function. This is the function that makes the actual call to pbi_makeport and does the plugin build.

${pluginroot}/transmission_pbi/

This is the Transmission PBI directory. All plugins must have a PBI directory. Within this directory, two subdirectories must exist: scripts/ and resources/. A pbi.conf file must also exist.

${transmission_pbi}/pbi.conf

This file tells pbi-manager how to build Transmission. It contains information about the plugin such as the port(s) to be built, the icon(s) to be used, the make options for the binaries, etc.

${transmission_pbi}/resources/

This directory contains the bulk of the plugin interface. It can be structured however the plugin author chooses. Since Transmission uses django, the django application resides in the directory along with an assortment of other scripts and programs.

${transmission_pbi}/resources/control.py

This is the transmission fastcgi server control program. As discussed previously, this program has three responsibilities: to start the fastcgi server, stop the fastcgi server and report the status of the fastcgi server. The start and stop methods also start and stop the django web server application. The django application exports all the required hooks that FreeNAS requires to interface with the plugin. A wrapper script that calls control.py is also in this directory. This “control” wrapper is the only mandatory file that needs to be known by the base system.

${transmission_pbi}/transmission

This is the RC script for transmission that controls the daemon. It's just like any other RC script that FreeNAS uses.

${transmission_pbi}/resources/tweak-rcconf

The job of this script is to modify /etc/rc.conf to enable or disable transmission.

${transmission_pbi}/scripts/

This directory contains hooks for different stages of the PBI build process and installation process. The possible scripts are pre-install.sh, post-install.sh, pre-portmake.sh, post-portmake.sh and pre-remove.sh. Pre-install.sh allows you to do customizations to the system prior to the plugin being installed, such as adding users and groups. Post-install.sh is run immediately after the plugin is installed. Some typical post install operations are database initialization and migrations. Pre-portmake.sh and post-portmake.sh allow you to do operations before and after port compile. Pre-remove.sh is run prior to plugin removal. Operations typically done by pre-remove.sh are user and group removal.

The other files in ${transmission_pbi}/resources are default.png, freenas and transmissionUI. Default.png is the default icon for the PBI. Freenas is a file that contains the plugin api version. TransmissionUI is the django application.

${transmission_ui}/freenas/

This is the django application. Under this directory are the typical django model, form, view and url files. In the urls.py file, the exported methods that FreeNAS requires are very visible and demonstrate what needs to be made available for a FreeNAS plugin to be functional.

To build the transmission plugin, run the command: ${freenas}/build/do_build.sh -t plugins/transmission. This will first create a ${freenas}/sbin directory and install pbi-manager into it. When pbi_makeport is invoked, it will compile a FreeBSD 8.x world and install it into a temporary directory which will later be tarred up and saved for future compilations. Once a world directory is ready, the ports that are needed to compile the plugin get compiled and installed. Any provided scripts get ran and then the PBI is made and placed in ${freenas}/${plugin}/${arch}/${plugin}.pbi. The plugin build is complete at this point.

To test and verify transmission works, upload the Transmission plugin through the FreeNAS web interface as previously described. Navigate to Services->plugins->Transmission from the navtree. Click on it and there should be an edit screen. There are default values filled in already but these can be modified and saved. Save the configuration and go to Services->plugins->Transmission from the main interface and turn the slider to on. At this point you can grab any torrent file, place it in the directory specified in the watch directory and watch it get downloaded in the download directory. Success!

8 Conclusion

FreeNAS plugins allow FreeNAS to be extended in ways anyone sees fit. They are very powerful in their flexibility and allow plugin authors to make a FreeNAS system into everything from a multimedia server to a print server. The purpose of this paper is to explain the plugin architecture so that more developers and people knowledgeable enough can make more plugins Happy hacking!