The PBI format re-implemented for FreeBSD and PC-BSD

Kristofer Paul Moore

kris@pcbsd.org
PC-BSD Software / iXsystems

Abstract

The PBI format (**P**ush **B**utton **I**nstaller) has been the default package management system for PC-BSD going on 5+ years now. However as we looked to the future it became apparent that it was greatly needing an overhaul to both improve its functionality, and expand its usage outside the scope of just PC-BSD. Among the areas needing improvement were how it dealt with identical libraries between applications, the heavy requirements from being implemented in QT/KDE, and lack of a digital verification mechanism.

Starting in April of 2010, work began on re-implementing the PBI format to address these issues, and greatly expand upon its usefulness as a package management system for both PC-BSD and FreeBSD. From this work the **pbi-manager**[i] was born as a subset of command-line functionality for dealing with every aspect of PBIs, from building, installing, distribution and advanced management. The resulting format has been implemented 100% in shell, allowing it to run virtually unmodified on a fresh FreeBSD system, as well as be agnostic towards which desktop a particular user may be running in PC-BSD. Features such as digital signature verification, intelligent library sharing, repository management, **bsdiff** updating and others have already been implemented, along with improved QT4-based front-ends, which behave and look almost identical to the legacy format. The end result is a powerful package format which can be used for traditional FreeBSD users as well as PC-BSD running any window manager, or none for that matter.

Introduction

When PC-BSD was first introduced back in April of 2005, a key feature of it was the introduction of the PBI (**P**ush **B**utton **I**nstaller) system of package management. This format was a new concept in the open-source arena at that time, with applications no longer relying upon a complex chain of library dependencies and instead coming bundled with all the necessary files in a single package. This format has been in use for all subsequent versions of PC-BSD though the 8.x series, making package management a thing of simplicity. However the format has begun to show its age, and there are a number of key areas in which it could be improved.

First among these is a solution to the duplication of identical library files between PBIs. When programs are packaged in the PBI format, often they will make use of the same libraries, such as X11, GTK, QT and more. The PBI format allows programs to each use different and possibly incompatible versions of these libraries, however there are a lot of cases there the libraries at 100% identical between them. This results in library-duplication on both the disk, and in memory at runtime. The re-implemented PBI format now has a mechanism for dealing with this, which can reclaim this wasted space while still providing a degree of separation between different PBIs.

Another issue facing the legacy PBI format has been its original implementation. When the PBI format was introduced it was written mostly in QT4 with some KDE library usage. While this has been fine on PC-BSD through the 8.x series, which was based on KDE / QT, it presented a problem with PC-BSD 9.x, which will offer choices between window-managers. This legacy implementation also

prevented easy usage on FreeBSD, as a number of prerequisites must now first be met to use PBIs, which in most cases made it unfeasible. With the new implementation of the PBI format, and its management tools, this issue has been solved, allowing it to be run on a native FreeBSD system, and possibly ported to other BSDs as well.

In addition to these issues with the legacy format, there have been a number of requests for new features to make the format more useful going forward. One such feature is the ability to digitally "sign" a PBI file, allowing the user / manager to verify that the binary is from the correct publisher, and has not been compromised either from a malicious entity, or simply corrupted during transit. Another new feature is the ability to update a PBI with small binary diff patches, which can reduce an update download to less than 5% of its original size. Also missing has been the ability to easily manage a "repository" of PBIs, which can include building the PBI from a FreeBSD port, distributing, and managing application updates. In this paper we will take a in-depth look at the fundamentals of these legacy problems, along with how the PBI reimplementation address them and expands the format to make it more powerful for a wider audience.

The PBI concept at its core

While the re-implementation of the PBI format has greatly changed its inner workings, the core concept has remained intact. This concept is about finding a way to distribute binary packages, which arrive in a self-contained format, that are able to function without requiring messy dependency resolution across the entire system. In effect this concept is a departure from traditional open-source package management.

Throughout the open-source desktop arena, the norm has been to blur the line between applications and the operating system. At any given point the various packages installed on the system makeup the users "Desktop", but this is always in a state of flux. By simply performing an upgrade of some seemingly trivial top level application, such as Firefox, there is always the risk that in turn lower level libraries / applications may also need to be updated. If one of these library dependencies fails to update, or introduces new bugs and regressions is it possible that other applications, seemingly unrelated to the original upgrade target could be negatively effected. While package management systems have become better about trying to track and fix these issues, the underlying problem still exists and every package upgrade becomes a potential land-mine which can negatively effect a desktops functionality. A graphical representation of this can be seen in Figure 1 below.

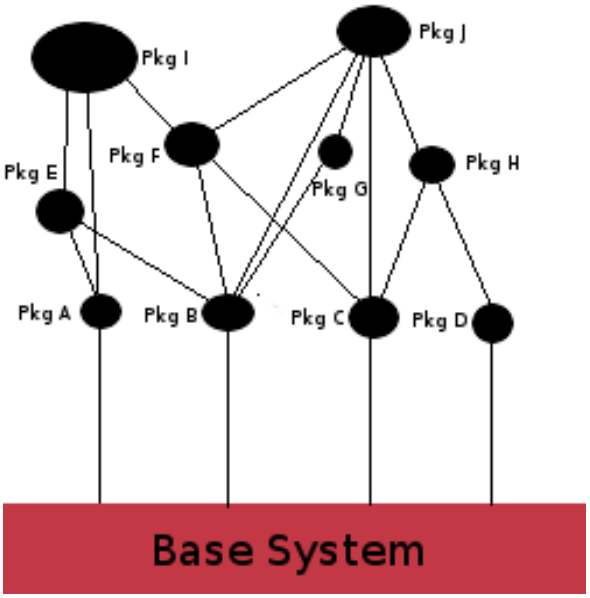

Figure 1: Traditional open-source packaging model.

Instead of simply writing another packaging system which follows this model, the PBI system rebuilds the walls between applications, and the base system. By packaging libraries and files together, an applications only fixed dependency is the base version of the operating system itself, so that a PBI built on FreeBSD 9.0 series should function on all minor versions in that branch, providing the ABI isn't changed at some point in the process. This may be somewhat of an oversimplification, since

some other prerequisites may still apply, such as X applications needing a running X server to communicate with, but from a library standpoint applications are fully functional. This approach to packaging eliminates the risks associated with updating or changing installed applications, since the target of the action can no longer modify the system libraries. It also provides a greater degree of freedom when building a desktop such as PC-BSD, which is able to distribute a fixed set of packages for a system's desktop environment, and let the user determine when they wish to upgrade their system, not simply be forced to simply because they wish to run a newer version of Firefox or some other application.

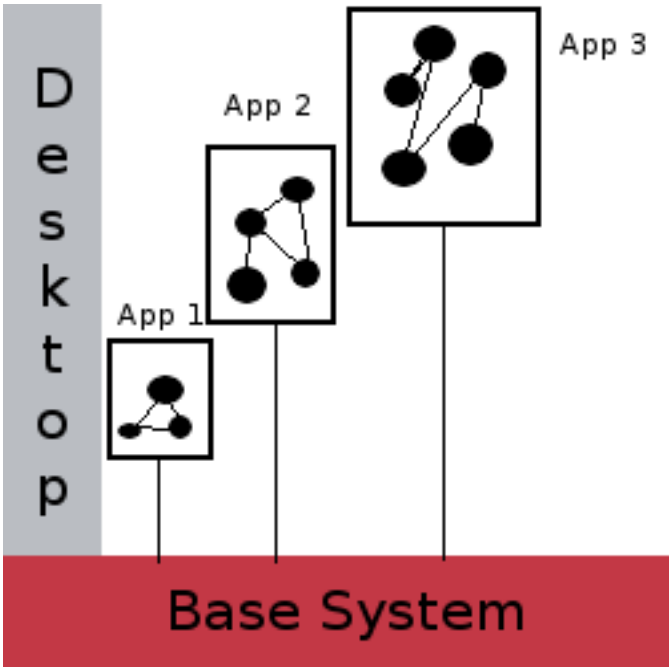

Figure 2: The separation between applications and the base system using the PBI format.

Intelligent sharing of libraries

While the bundling of applications together with the required dependencies has worked to eliminate the risk of installing / updating applications causing system breakage, it has also created a new issue of possible disk and memory bloat. In the old implementation of the PBI format, any group of applications may be including similar or often identical libraries between them. While disk-space is cheap enough to make this less of an issue, it is still wasteful and is more of an issue at application runtime, with each program possibly loading identical copies of a library into precious RAM space.

To solve this problem, the new PBI format has implemented a new system of sharing identical libraries via hard-links, in a central location we will refer to as the “hash-directory”. By using this technique we are able to solve both problems at the same time. First by using hard-links we are able to reclaim lost disk-space since we only have a single copy of a particular library version. In addition the system will no longer need to load the same library into memory at runtime, since the inode will match for each file record.

The mechanism for using this new hash directory is first started when a PBI is created. When the PBI is built, the **pbi-manager** will search through the common library directories and build an index of the files, along with their unique sha256 checksums. During this process, files will be inspected for obvious unique strings, such as dates or prefixes, and pruned from the index, to help ensure that files in the hash-directory are truly sharable. This index is included in the final PBI file and upon installation will be marked as ready for inclusion to the hash-directory. From this point the work is passed off to a new “**pbid**” daemon, which tracks the installation / removal of PBIs, is able to transparently merge libraries into the tree and remove stale entries.

Within the hash-directory, files are named according to the base filename, with the sha256 checksum appended to the end. The **pbid** daemon is able to quickly parse the pending file indexes, looking for matches of filename and checksum in the hash-directory. When a match is found, the existing file in the PBI's directory will be removed, and a hard-link created in its place. Should the file not already exist in the hash-directory, it will be created and hard-linked back to the original location within

the PBI. This means in effect, that as PBIs are installed and the hash-directory grows, there will be greater likelihood that a file will already exist in the hash-directory, and the **pbid** program can safely reclaim that space, substituting a hard-link in its place.

Going in the opposite direction, during the removal of a PBI and its files, the hash-directory will be marked as “dirty”. When the **pbid** daemon encounters this condition it will then do a quick walk through of the hash-directory, looking at the reference count for each file. When the reference counter for a file has dropped to 1, that indicates that it is not being used elsewhere and is now safe for removal from the hash-directory.

Implementation and Requirements

Another issue which had to be corrected for the new PBI implementation was that of its own requirements. Historically the PBI format was written in C++, using a variety of QT and KDE libraries. While this was a workable solution for PC-BSD up to the 8.x series, it made it very difficult to use PBIs on traditional FreeBSD and the upcoming PC-BSD 9 series, which no longer provides KDE standard on all installations. Each PBI also contained the installer mechanism built-in to the archive file, which made it difficult to maintain backwards compatibility when moving to systems which were built upon newer revisions of QT/KDE.

With this problem in mind it made much more sense to implement the PBI tool-chain in a way which did not force the end-user to have a particular version of a library(s). Using an interpreted language, such as shell, and only relying on FreeBSD base system commands was the logical choice, completely removing the PBI system from a clutter of its own dependencies.

This is an important choice for several reasons. First, it makes the PBI format useful as a binary package management system, on an out-of-box FreeBSD installation, even without having X installed. Second it allows desktop users to utilize the PBI format, agnostic towards whatever particular desktop manager they have chosen to run, as we have done with PC-BSD 9. Also by using the freedesktop.org XDG specifications[ii] for desktop icons, mime types, and menu entries, the PBI re-implementation is now able to manage the installation and removal of these for a variety of XDG compliant desktops.

Command-line functionality

A related weakness to the QT/KDE implementation of the legacy PBI format was its over-reliance upon GUI tools and utilities for PBI installation and management. While a few basic command-line utilities were provided after the initial release of the PBI format, they still could only interact with a very basic sub-set of PBI features. Among others, there was no command-line line ability to update or search for PBIs to install, which greatly hindered the usefulness of the original implementation in a native FreeBSD environment.

With the reimplementation of the PBI format being 100% in shell, this has allowed all functionality to now be utilized directly from the command-line. In PC-BSD specifically, any GUI tools are simply front-ends to their command-line equivalents. The shell code of the new PBI tools is entirely contained within a single file, named **pbi-manager**[iii], however due to the large scope of the project, a number of commands have been created to access particular features of the script. In order to minimize the learning curve for these commands some names and flags have been copied from FreeBSD's built-in package utilities. Lets take a closer look at some of these new commands:

pbi_add

This command is used to perform a number of actions relating to adding or installing a PBI on a system. Some important flags for it are:

-i Display PBI build information, digital

signature and more.

--checkscript Display any installation / removal scripts included with this PBI

-r Fetch the PBI file and install from a remote archive

pbi_info

The **pbi_info** command can be used to display information about the PBIs installed on the system. It also provides a '**-i**' flag, which can be used to display a listing of available PBIs from it's known repository index files

pbi_update

Checking for updates, and updating a/all PBIs is performed through the **pbi_update** command. The command provides options to generate a listing of all available updates, automatically update PBIs, or update a specific application.

Digital Verification of PBIs

One often requested feature for PBIs has been the ability to digitally “sign” and then verify these signatures before installation. This can be used to ensure that the PBI contents have not been corrupted or tampered with from the time of building, until actual installation on the end-users system. This feature is standard in the new PBI format, and also an integral part of how we link a specific PBI to a target repository.

For digital signing of files, the PBI system uses the **openssl**[iv] command-line utility, which is included in the FreeBSD base system. This command can be used to generate the private / public keypair, as well as perform the signing and verification process. After creating a new set of private / public keys, the private key can be used with the **pbi_makeport** and **pbi_create** commands to sign critical portions of the resulting PBI file. Any installation and removal scripts are signed, as well as the checksum of the PBI archive tar-ball. These files and signatures are then stored in the header of a PBI file, which allows quick verification of signatures later on. The public key is used for verification of signatures, and is tied into the repository management, which we will be taking a closer look at in the next section.

PBI Distribution and Repositories

Another new feature added by the PBI reimplementation is the framework for distributing and updating PBIs through the use of PBI repositories. With the **pbi_makerepo** command, it is possible to create a small repository file (.rpo), which can be distributed to end users for registration of the repository on their system. A repository is made up of several key values:

- Description
- Public Key
- Mirror URL
- Index URL

The description field is what most end-users will see when looking at their list of installed repositories, such as “PC-BSD PBI Repository”. The public key is from the **openssl** key pair created and being used to sign PBIs made for this repository. The Mirror URL would be the master URL for downloading PBI files and updates. Lastly the Index URL is used to specify the location of the repository INDEX file, which we will take a further look into below.

A repository INDEX is a file the **pbi-manager** uses in a variety of tasks, such as checking for updates, searching for new PBIs to add, and providing listings of available PBIs on a repository. Normally it is automatically downloaded by the **pbid** daemon on a daily basis and used mainly as a way to stay updated with the latest versions of applications. Within the index are a variety of fields for PBIs, such as names, versions, checksums, architectures and more. Repository managers are also provided with a command **pbi_indextool** which lets them

easily add / remove entries from their INDEX for publishing. Using the repository tools, along with commands for building PBIs, it is now possible to run an entire distribution network from a native FreeBSD system with no ports installed.

Binary Patching / Updating

Over the past several years of using the legacy PBI format there have been many requests for a way to shrink the amount of data downloaded for updating. Typically whenever a PBI update was issued, it would require the client to re-download the entire archive for installation, which due to the dependencies being included, can quickly grow to a non-trival size. A number of solutions have been experimented with during the development of the new PBI format and the current solution of using binary patching through the **bsdiff**[v] / **bspatch** commands was implemented. Instead of requiring the re-download of a full PBI file, updating can now be performed through a much smaller binary patch file (.pbp). In some cases this patch file may be less than .05% of the original PBI size! This patch file contains an archive of binary differences produced by the **bsdiff** command, updated hash-directory listings, removed files list, and new files. The resulting .pbp file can also be digitally signed to ensure the legitimacy of the contents. In order to support this method of binary patching, a couple of new command-line utilities have been created:

pbi_makepatch

This command is used in the creation of a new **P**ush **B**utton **P**atch (.pbp) file. In requires two PBI files, both of the same application and architecture but different versions. Both of the PBI files will be unpacked and the contents of them will be inspected. Any files which have been removed or added to the newer PBI will be recorded, and then each file will be inspected to determine if it has changed in any way between the versions. If the file has been modified, then **bsdiff** is run to generate a differences patch between the two. These patch files are then saved, along with some header information into the resulting .pbp file.

pbi_patch

This command is used for applying a PBI binary patch file (.pbp) to upgrade an installed application to a newer version. It has a number of flags similar to **pbi_add**, for viewing information about the PBI, checking scripts, digital signatures and more. When starting the patching process, some basic checks are first done to ensure this patch is intended for the version of the PBI installed on disk. If these checks pass, then the patch is extracted, and files updated using the **bspatch** command. During this process checks are performed on each file, and if it is hard-linked to a shared file in the hash-directory, then steps are taken to unlink and update the file, before trying to merge it back into the hash-directory.

Building a PBI

In the previous iteration of the PBI format, a couple of separate tools were available which handled the creation aspect of applications. The 'PBI Builder' provided a mechanism for compiling ports into workable PBIs via the command-line. Another application, the 'PBI Creator' was also available, which provided a GUI for generating simple PBI's without handling the port building structure.

The functionality of both of these tools have been built directly into the **pbi-manager**, via the **pbi_autobuild**, **pbi_makeport** and **pbi_create** commands. By having these commands built in, it is easy for any person to start building PBIs, or even run large batch builds to convert FreeBSD ports into PBI files. Let us take a closer look at these commands and their specific functionality.

pbi_makeport

Administrators and users wanting to build their own PBI files from a FreeBSD port

will find **pbi_makeport** command very useful. By using this single command, it is possible to perform a fresh build of a target port, its related dependencies and bundle them together into a final PBI file. Upon the first run of the command, it will start by building a fresh buildworld / sandbox environment from the appropriate FreeBSD sources. Once this process is finished it will be saved for future builds, and extracted into a clean chroot environment in which to perform the various port make steps.

Because each PBI is built with a unique PREFIX, a couple of special actions are taken within the clean chroot environment before starting the port compiles. First a couple of make options are set to automate the build, and specify a correct PBI prefix:

BATCH=yes
PREFIX=/usr/local/pbi/<appname>-<arch>

In addition to setting make options, the new prefix directory is created as a symbolic link back to /usr/local. This step is important, since many ports in the FreeBSD ports tree do not like to be compiled out of their default /usr/local prefix, but will still function properly at runtime. At this point the build is started of the target port, which builds it and all its dependencies unique to this PREFIX. If the port build finishes successfully then the package and its dependencies are analyzed and any build-specific ports are removed at this point.

Now the PBI creation process is started, first by analyzing libraries / files for potential hash-directory candidates and generating a list of matches. Next the target ports file-listing is used to identify executables, libraries, and other files which will need to be sym-linked back into the /usr/local directory at installation. This step is important to ensure that executables end up in the users PATH and libraries can be found by other programs. Finally the entire PREFIX directory is saved into a compressed **tar** archive, PBI information is saved into a header file, digital signatures are created and the final .pbi file is produced.

pbi_autobuild

The **pbi_autobuild** command provides an easy to use framework for maintaining a tree of PBIs which are automatically rebuilt when the underlying FreeBSD port is updated or when the administrator manually bumps the rebuild trigger. It provides flags to enable a variety of options such as digital key signing, automatic creation of binary patch files for new PBIs, pruning and more.

pbi_create

The **pbi_create** command can be used to manually create a PBI file from a variety of sources. It is similar to **pbi_makeport**, in that it creates a PBI file from a target directory, but without the port building magic. It can be used as a way to manually build PBI files without using an underlying ports infrastructure and still end up with a valid PBI which is digitally signed. In addition it also supports re-packaging of an already installed application, back to a PBI file.

Conclusion

The re-implementation of the PBI format has preserved many of the core concepts present in the original specification. It still provides a binary package system, which eliminates the need to maintain messy dependency tables. We have taken a look at some of the specific improvements to help solve the disk and RAM bloat from duplication of identical files, digital verification, repository management and more. By switching the PBI tool-chain to a shell-based environment the doors have been flung open to allow all FreeBSD users access to using PBIs on their desktop and servers. For the release of PC-BSD 9.0 later in 2011 this reimplementation will become the default, allowing users to run a variety of different window-managers and still maintain a cohesive package management experience.

i http://wiki.pcbsd.org/index.php/PBI_Manager
ii http://www.freedesktop.org/wiki/Specifications/desktop-entry-spec
iii http://trac.pcbsd.org/browser/pcbsd/current/src-sh/pbi-manager/
iv http://www.openssl.org/
v http://www.freebsd.org/cgi/man.cgi?query=bsdiff&apropos=0&sektion=0&manpath=FreeBSD+8.1-RELEASE&format=html