

# Gentoo for all the Unusual reasons

*“How a source-based Linux distribution nicely solves the problem dealing with newer versions of software”*

## Introduction

I have a confession to make. I use Gentoo Linux. All my colleagues at the various Linux User Groups I attend think I'm nuts.

“Everyone knows” that Gentoo is a “source based” Linux distribution. Gentoo's reputation (in large measure pushed by the people who develop the distribution) is that it's for people that want super crazy optimizations, and really only suitable for those who use desktops.

It turns out that Gentoo is really ideal for a whole bunch of other, unexpected, reasons - and that, much to my surprise, there are actually people using Gentoo in production environments for these very reasons.

## Speed

Because there is binary compatibility across all the descendants of the original i386 processor, the other Linux distributions (not to mention everyone writing software for Microsoft Windows) can just ship prepackaged binary versions of their software compiled for the generic i386 platform and take advantage of the fact that it'll work everywhere.

That does mean, however, that they are unable to take advantage of any new optimizations that your fancy [expensive] CPU might offer, which is a pity.

Since Gentoo is a built-from-source distribution, you are able to specify compiler flags to be used when building software for your system. `gcc` in particular allows one to specify what kind of CPU you're going to build the code for. By specifying the processor type (Intel Pentium III, AMD Athlon Thunderbird, Sun UltraSPARC, etc), the compiler is able to generate processor-specific code which (in theory) will result in better (ie hopefully faster) machine code.

Is a Gentoo system faster? Anecdotal evidence is mixed. It seems to work out that Gentoo system will run somewhat faster than an identically configured one from one of the more popular distributions, but any minor performance advantage will be completely squandered if the system is not installed, configured and tuned correctly. Since many of us don't know how to do that, and since Gentoo does offer so much latitude to do your own thing, it's easy to lose the benefits of slightly faster programs if you do something silly.

So, it seems from a speed perspective it really doesn't matter whether you use a build-it-from-source distribution or a binary-package distribution.

So if that's not a reason to use Gentoo, why would you want this built-from-source thing?

## Common problems in production environments

There are two related reasons why people start getting annoyed at their computers. [Actually, come to think of it, there are a whole galaxy of reasons why people get annoyed at computers, but I'll focus on just these for today]. I call it “The newer version problem”, and there are two ways that modern operating systems run into it:

### **The newer version problem (1) – what if I need something the OS doesn't provide?**

Every distribution (be it Linux or Unix) faces a similar problem in the real world. It doesn't matter which one. The vendor provides the operating system, the GNU toolchain that we all depend on, and tons of other software that gives us a fantastic computing environment. Inevitably, however,

there is something that we need that the distro doesn't provide.

This is a crucial point. Invariably, there is something you're going to have to roll out on your own. Two examples:

On the server: you've developed an application which relies upon a new feature in the latest PHP, version 4.3.4. Unfortunately the version of PHP packaged with your Debian system, 4.1.2, doesn't have the functionality you need.

On the desktop: perhaps you're doing some graphics work on your Red Hat workstation, and want to take advantage of the new soft-ray tracing that the latest `blender` has, and they don't provide it for you. Whatever your particular need is, if you want it you're going to have to build and install yourself.

So if upgrading when you choose to is a task you're going to have to do, the real question is "does the operating system help you do it?"

## **The newer version problem (2) – hey, they fixed something! Why can't I have it?**

Let's face it – free software isn't always perfect. (I know, big secret, wasn't supposed to tell anyone). Commercial software isn't perfect either, but there's a crucial difference: *sometimes* (and especially in the case of major software like an operating system or a database - I'm thinking Solaris and Oracle here) the vendor will support older versions. If I'm running Solaris 7, and there's a critical driver bug or security issue, then Sun will probably issue a patch that I can use to upgrade my systems.

It's not like that in the free software world. With Apache and the Linux kernel itself being about the only exceptions, no one supports older versions of their software. Any time there's a bug fix or a usability enhancement it's in the next version.

The current release version of Debian Linux, “stable”, as it is known, is renowned for its robustness and its ability to run in a wide variety of environments. They achieve this by freezing the versions of various packages at a particular point, then extensively testing that everything works together. Unfortunately, while the core system libraries are rock solid as a result, the user applications (say, a graphical editor or mail client) and common system applications (like the aforementioned PHP) are also frozen in time. Take `evolution`, Ximian's outstanding email client. Debian stable has version 1.0.5. Evo, on the other hand, has moved through an entire 1.2 stable release, and is now well into a *second* stable release – [at time of writing] the current version of evolution is 1.4.5. When someone running Debian stable (and hence evo version 1.0.5) contacts the evolution user mailing list asking for help, she is invariably told that the solution to her problem can be obtained by “upgrading to the latest version”. Debian stable, however, doesn't help her do that.

## **Help me out here!**

There's a key issue at play: ease of operation. Does the operating system help you with the challenges that everyday life administering a system presents? Much to my surprise, Gentoo Linux turns out to be really good in this regard.

With Gentoo, one installs new packages by downloading sources and then compiling them. You want a piece of software, no problem. Just issue the instruction, and away it goes. A little while later, it's installed. It's the same user experience as with Debian. But when I'm faced with the need to get a newer version of a piece of software - that's when Gentoo shines.

Here's an example: let's say I'm using `bluefish`, the HTML editor, and there's a bug that's

annoying me. First of all, a newer version might be available in Portage (Gentoo's software package management system) so I might be able to just ask for the upgrade. Gentoo has a handy command called `etcat` which is one of several ways I can tell what's available:

```
# etcat versions bluefish
```

```
* app-editors/bluefish :
  [ I ] app-editors/bluefish-0.9 (0)
  [   ] app-editors/bluefish-0.11 (0)
  [   ] app-editors/bluefish-0.12 (0)
  [M~ ] app-editors/bluefish-0.13 (0)
```

This tells me I've got version 0.9 of bluefish installed, and now there's a 0.12 available. From reading their website, I know that they've fixed the problem I'm having, so I definitely want the upgrade!

The `emerge` command can tell us what will happen if I do upgrade:

```
# emerge --pretend bluefish
```

```
These are the packages that I would merge, in order:
Calculating dependencies ...done!
[ebuild N   ] dev-libs/libpcre-4.2-r1
[ebuild   U ] app-editors/bluefish-0.12 [0.9]
```

Apparently this version of bluefish needs a library called `libpcre`. Portage has shown me that in addition to doing an upgrade of bluefish, it's going to bring in `libpcre` as well. Hey, fine with me. So off we go:

```
# emerge bluefish
```

First Portage downloads, builds and installs `libpcre`, and then it does the same for bluefish. Four minutes later, I had my upgrade. Pretty easy.

You might have noticed that it didn't say it was going to install version 0.13. That's because, at present, it's “masked” (that's why it showed up in red). In this scenario, 0.13 just came out, and there's now an ebuild for it. The ebuild, though, is still being tested to see that the software actually installs and that there's nothing blatantly wrong with it.

Maybe it'll be “unmasked” tomorrow, maybe next week, and maybe never if it turns out to have problems and is superseded by an even newer version. If I had really needed it, I could have overridden Portage and told it to bring in 0.13 in. Likewise, I could have picked version 0.11 if I'd had a reason to. In my case, I knew my issue was fixed in 0.12, the most recent available, and so I just let Portage do its thing.

This flexibility is one of Gentoo's greatest strengths.

## Do it yourself packages

A trickier situation occurs when I need to install a piece of software that the system doesn't provide.

One of the significant reasons why various distributions evolved package management tools was so there would be a single, unified view about what is installed on the system. For each piece of software (be it a basic system tool, a core library, a server program, or a user application) a **package** is made. As each is installed on your system, the OS records what files got put where, and that the package is installed. That way other software which depends on that package can be

installed knowing that their prerequisite pieces are in place.

But what happens if you install a newer version of software and don't have a package appropriate to your OS? You typically go through the same build steps that the person who built the package did, only you probably either:

- a) install it in some private place, perhaps `/usr/local/bin`, and then go to the effort of making sure "your" program is getting run (not the older one); or
- b) blindly install your software to the root filesystem, hoping that you don't clobber anything on the way in, and praying that nothing in the future will ever overwrite the programs and files you have installed.

Think about that for a minute. Doesn't having to worry about these things strike you as a bit silly? After all, isn't that what the package management system is supposed to prevent?

The usual answer at this point is "well, just create your own package from the software you want to install". Fair enough.

## How easy is that to do?

The question I'm posing isn't "does the ability to make packages exist" (because of course the answer to that is yes across the board), nor am I asking "can you create your own packages", but rather "how easy is it to do so"?

Lets say you've got the OS provided copy of `bogofilter`, and let's say you've got an `.rpm` for version 0.16.1 . All good. But suddenly the authors of `bogofilter` discovered that there was a silly (but serious) error that crept in, and so a short time later released 0.16.2

That upstream has fixed the problem is evident - but this glitch happens to be affecting you directly so you want it done away with. The problem is that you're stuck with waiting for your distribution to release a new version of the `.rpm` (or `.deb`, or `.pkg`, or ...), and that could take a very long while. Which leaves you in the position of wanting to roll your own.

That's where the trouble creeps in. Conceptually, just creating your own new `.rpm` or `.deb` package is easy, right? "Just use the existing 0.16.1 package as a prototype." But for most people (ie anyone not at wizard level and sometimes not even then) it's actually rather tough to do: You have to:

- download the package description or somehow extract it from the existing package file;
- manually download the new version of the upstream `.tar.gz` (or whatever) source and unpack it;
- transplant the build descriptions (in the case of Debian into the new upstream sources) and maybe even patch against those sources;
- you might have to modify the build script to instruct it about the new version;
- actually try and create the package, which of course involves compiling it (which probably also will require you to install a large number of "-dev" packages you hadn't previously known about - real pain);
- then you install, and test.

While this is all doable there's a fairly steep learning curve (especially for newbies) in getting the skills needed here. More to the point, it's a lot of work that you'd rather not do.

## Enter Gentoo

Conceptually no different than above, on a Gentoo system it's just easier.

The magical part is that package description files in Gentoo – “ebuilds” – follow a very simple format. They're basically shell scripts (see sidebar). Along the way you specify where to get the source tarball from. When you build, Portage downloads the source for you and then proceeds to unpack and compile it. Because they're shell scripts, they can use shell variables to great effect; in particular, they take the version number by parsing the ebuild filename and putting it in a variable the script can use.

In our bogofilter example above, the package file (called bogofilter-0.16.1.ebuild) contains a line like this:

```
SRC_URI="http://sourceforge.net/downloads/bogofilter-${PV}.tar.gz"
```

When you go to build and install bogofilter, Portage sets \$PV to be 0.16.1 based on the filename, and fetches the appropriate .tar.gz. It then unpacks it, and proceeds to ./configure; make; make install and then build the package as instructed. So, guess how hard it is to create an ebuild script for the new version you want, 0.16.2?

```
# cd /usr/portage/net-mail/  
# cp bogofilter-0.16.1.ebuild bogofilter-0.16.2.ebuild  
# ebuild bogofilter-0.16.2.ebuild digest
```

Done!

Assuming that nothing in the package description, unpacking instructions, etc, needs to be updated, that's it. Certainly for a minor version change, you're usually you're all right.

There's a touch more to keep abreast of – for example, you probably would do the above action in a private copy of the /usr/portage tree (say, /usr/local/portage), so you don't lose your changes when the primary tree updates. Portage explicitly supports this - just look in the description of the PORTDIR\_OVERLAY variable in the online documentation or right in /etc/make.conf to learn how to tell Portage where your custom ebuilds are.

Now you can just tell Portage to:

```
# emerge bogofilter
```

and you'll have your new version!

Gentoo has copies of the source tarballs required for all of its various packages on its mirrors around the world. So, normally Portage just gets the source from one of them. If, however, you're building something which isn't in Gentoo's mirrors, no problem – Portage will just reach out to the original upstream download site.

Portage uses md5sums to help ensure that you get an uncorrupted downloads. That's what the third command above, ebuild ... digest, was for - it downloads the source and then computes the md5sum for you. Since you're the one doing the version bump, (instead of relying upon one made by the Gentoo Developer team) its up to you to make sure you actually did get an uncorrupted download, so really you'd probably do ebuild ... unpack first to get the download and make sure it looked ok, then do the digest command once you're happy.

Finally, if you want software for which your OS doesn't provide a package, you of course have to write your own. With Gentoo, writing a custom .ebuild is easy – see sidebar.

## Administering multiple machines

Consider these problems not just on a single desktop, but in the context of a production platform of dozens of servers or thousands of workstations.

Frankly, there aren't any Operating Systems out there that give you much help here. There's an entire body of literature on the subject of infrastructure management; sadly, there's still a great deal of ad-hoc deployment that goes on. While many vendors have tools that help you build a series of systems the first time, the task of maintaining them over time is left to the individual site to deal with. The newer version problem isn't just about single machines – it's about entire networks of them.

So how does Gentoo stack up in production environments? Another surprise for you from the source based distro: Portage can be told to build binary packages. This allows you to have one machine over in the corner doing all the compilation work, and then the packages can be shared out and used by all your target machines (instead of them having to build the packages themselves). You might be tempted to say “isn't that what the other Linux distributions do?” The difference is that selecting the right mix of packages is a site decision, and the newer version problem is definitely a site burden to deal with. Gentoo gives the local systems team the tools to deal with solving these version issues themselves.

By using a local build server you can concentrate horsepower and version management effectively, yet still have room for local customization. Staging environments are easy to set up. Then, once you're happy with the set of versions you've tested then you just make a snapshot of those binary packages and share them out to your rank-and-file machines. Recent versions of Portage include built in support for fetching binary packages you've created from local file servers, so all this works, right out of the box.

## **Conclusion**

Create your own package, or privately version bump an existing one – the newer version problem comes up all the time. The more mainstream package management tools, while "mature", require a much greater level of effort to accomplish these tasks. Conceptually, though, the tasks are trivial. Quite to my surprise (since they don't run around advertising this aspect), the design of Gentoo's tools makes it really easy to do these things yourself.

## **Use the source, Luke**

As I watch packages build, I'm awed again and again at the vast contribution made by so many in the Free Software world. The code really is right there, and that makes it that much easier to make a change and maybe contribute back.

And there's just something nice about using Open Source software that I can see coming *from* the source.

Unusual reasons indeed.

## Acknowledgments

The author would like to thank Stephen White (University of Adelaide) and Andrea Barisani (University of Trieste and Gentoo developer) for having helped develop the ideas here as they relate to production use of Gentoo. They also kindly reviewed the article, as did Pia Smith (Linux Australia), Jeff Waugh (GNOME), Craig McWhirter (SYDNEY LINUX USERS GROUP), and Wade Mealing (Gentoo server project).

## Author



Andrew Cowie runs [Operational Dynamics](#), an operations and infrastructure engineering consultancy. He helps organizations get value from their technology, but does so by focusing on people and the processes around people – which is probably why he's so obsessed with finding easier ways to do things. You can reach him at [andrew@operationaldynamics.com](mailto:andrew@operationaldynamics.com) or as AfC on irc.freenode.net

## All about ebuilds (a sidebar)

Gentoo's package descriptions are “written” in `bash`. They are basically just shell scripts! The various instructions go in functions which get called by Portage at various points along the way. The major ones are:

```
pkg_setup()
src_unpack()
src_compile()
src_install()
pkg_preinst()
pkg_postinst()
```

which get called in order. To tell Portage how to build your software, you just write functions for each of the steps, and proceed them with a bit of information (like the `SRC_URI` discussed in the main text).

To compile your sources, you might have

```
src_compile () {
    ./configure --prefix=/usr
    make
}
```

The amazing thing about the shell script thing is that by overloading functions, they can provide sensible defaults. In fact, the default for `src_compile()` is pretty much what I showed above. And for many packages, that's perfect. In fact, you could write an ebuild which relies on the defaults and has no custom functions defined at all. It works quite frequently! Talk about easy.

Sometime you want to `./configure` a package differently depending on what sort of system you have. Portage has an environment variable called `USE` (set in `/etc/make.conf` and overrideable on the command line) which contains tokens you can use to describe and customize your system. Say you've got a package that can be told to build differently depending on whether or not you want, say, X Windows support, or IPv6 support. Your `src_compile()` function might look like something like this:

```
src_compile () {
    use X      && conf="${conf} --with-x"
    use ipv6  || conf="${conf} --without-ipv6"

    ./configure --prefix=/usr ${conf}
    make
}
```

You can see various features of shell scripting being used. In this example, if your system (like most users' machines) has X windows on it, then this package will be told to go ahead and build X support in. But if it's a server, and you don't need any of that, then your software gets built without that extra overhead. The `USE` variables can be overridden on the command line, so you have even more precise control if you need it!

`src_unpack()` is the same. If you don't include one, Portage will just plow ahead, untar the source tarball in the default place, `chdir` to that directory, and set the working directory environment variable, `$WORKDIR` accordingly. On the other hand, if something unusual has to happen (say, a patch be applied) then you can write a simple `unpack` function yourself; they provide some really useful tools to help things along:

```
src_unpack () {
    unpack ${A}
    epatch ${FILESDIR}/fixit.patch
}
```

So, with knowledge of the default activities, the assumed working directories, and the automatically set environment variables, you have enormous power at your fingertips.

I'll conclude with a full example. I had a client which exclusively used ssh.com's implementation of the SSH2 protocol. So, I needed to install it on a number of machines. See Listing 1, "ssh2-3.2.9.1.ebuild"

An ebuild starts by setting a number of environment variables, including

**SLOT** Typically used for libraries. When an ebuild author knows multiple [major] versions of the same packages can be installed on a system at the same time they can assign a slot number to distinguish them; On one of my systems I have Berkley DB version 1.85 (SLOT 1), version 3.2.9 (SLOT 3), version 4.0.14 (SLOT 4), and even version 4.1.25\_p1 (SLOT 4.1) - there is plenty of software out there that was written to use the older APIs; no reason they shouldn't be able to be installed, and that's why there are so many different versions installed.

If a newer version in the 4.0 series is released as stable, say version 4.0.17, then as long as it is still in SLOT 4 then my system will offer me the upgrade from 4.0.14, without removing the other versions installed.

Admittedly, Berkley DB is one of the more complicated examples out there, but it demonstrates the power behind Gentoo's slot implementation.

Most ebuilds don't need any of this, and just say SLOT="0"

**KEYWORDS** This is you indicate support for different architectures. In the example, I've shown that this ebuild is known as working and stable on x86 series platforms. The ~ in ~ppc means that it's "masked" - I know previous versions build on Power PC systems, but I don't have one handy to test with so others may want to take caution before deciding to install this version. In the official Portage tree, an ebuild like this would stay in this state for a few weeks until a people using Power PC were able to test the ebuild; after several positive reports the ebuild would be "unmasked" to ppc.

**DEPEND , RDEPEND** This is where dependencies are listed. It's a fairly complete grammar; particular versions of necessary packages can be listed. The most common modifiers are >= (to indicate that at least that version must be installed, usually because of an API that our program depends on) and ! (used to show that this package conflicts with the presence of another, and so tell Portage that it can't install both at the same time).

RDEPEND is for run time dependencies - things that have to be installed to be able to use package, whereas DEPEND are dependencies to build it in the first place; the difference only shows up when you're installing binary packages that were built elsewhere.

RESTRICT      Various fine grained controls of Portage's features are possible. In this case, since this is an ebuild I cooked up myself, I use `nomirror` to tell emerge not to bother looking in Gentoo's family of mirrors and instead go straight to the upstream provider. Note that that doesn't actually imply I can't use a mirror provided by the upstream authors; in fact, if you look at `SRC_URI` you'll see that I've listed a mirror very close to me where I know I should be able to get the `.tar.gz` I need.

Then we proceed onto overloading the various functions that control how the package is built.

The `src_compile()` function is the interesting bit. I've taken the example above, and fleshed it out a bit. You can see that some options are controlled by `USE` variables, while others we just go ahead and specify, like where we want the configuration files to go.

We don't really need the die failure messages, but they illustrate how we have full semantics and power of a shell script available.

Finally, in the `src_install()` function, we could have relied on the default, but on my system, files in `/etc/init.d` don't have `".rc"` appended to them. More importantly, since this is intended to replace (on the target systems where this ebuild was deployed) `openssh`, I wanted to be clear that the `rc` script was different than the one that `openssh` put in place.

Portage provides a rich library of helper functions which help simplify doing common tasks. We take advantage of one to say where we want the `rc` script to go and to see that it is marked executable.

And that's it! You place the ebuild into your local overlay of the Portage tree, and tell emerge to do its thing.

This really only scratched the surface. For more details, see

- [http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc\\_chap2](http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc_chap2)
- the output of `emerge --help`
- the man pages for `ebuild(1)` and `ebuild(5)` on any Gentoo system

## Listing 1: ssh2-3.2.9.1.ebuild

```
DESCRIPTION="ssh.com's implentation of SSH2"
SRC_URI="
ftp://mirror.aarnet.edu.au/pub/ssh/ssh-${PV}.tar.gz
ftp://ftp.ssh.com/pub/ssh/ssh-${PV}.tar.gz"
HOMEPAGE="http://www.ssh.com/products"

SLOT="0"
LICENCE="free-noncomm"
KEYWORDS="x86 ~ppc"

RDEPEND="virtual/glibc
!net-misc/openssh
>=sys-libs/zlib-1.1.4"

DEPEND="${RDEPEND}
dev-lang/perl
>=sys-apps/sed-4"

PROVIDE="virtual/ssh"
IUSE="X ipv6"
RESTRICT="nomirror"

# we're calling the package ssh2; the source
# tarballs are all ssh-x.y.z So, we have to
# overwrite S to specify the actual name of the
# directory as unpacked

S="${WORKDIR}/ssh-${PV}"

# probably could have relied on the default here

src_unpack() {
    unpack ${A}
}

# Of the large number of configure options that
# are offered, we offer customization of
# whether X windows and IPv6 support are
# compiled in.

src_compile() {
    local conf

    use X    && conf="${conf} --with-x"
    use ipv6 || conf="${conf} --without-ipv6"

    ./configure ${conf} --host="${CHOST}" \
        --prefix="/usr" \
        --with-ssh-agent1-compat \
        --with-etcdir="/etc/ssh2" || die "configuration failed"

    make || die "compile failed"
}

# again, almost the default pattern, but
# we want to change the name of the rc script

src_install() {
    make DESTDIR=${D} install

    exeinto /etc/init.d
    newexe ${FILESDIR}/sshd2.rc sshd2
}

```