# Reproducible Builds in FreeBSD

Ed Maste
The FreeBSD Foundation / The Linux Foundation
Email: emaste@freebsd.org

February 3, 2017

## Abstract

The goal of a reproducible build is to allow anyone to build an identical copy of a software package from given source code, to verify that no flaws have been introduced in the compilation process. This paper presents an introduction to reproducible builds, explains why build reproducibility is desirable, reports on the current state of build reproducibility in FreeBSD, and examines some of the techniques used to obtain reproducible builds.

## 1 Introduction

From the Reproducible Builds definition[1],

> Reproducible Builds are a set of software development practices that provide a verifiable path from human-readable source code to binary code executed by a computer.
>
> A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.

Developers of many projects, including FreeBSD, have been working for several years on making their builds reproducible. There are different motivations for doing so. Reproducible builds provide security and software assurance benefits including the detection of malfeasance, support the goals of the Free Software community, and provide operational and efficiency developmental benefits unrelated to security.

### 1.1 Security and Software Integrity

One benefit of Free and Open Source software is that the source code is publicly available for audit and review. If we assume that such review occurs, and that the tools used to build the software are trustworthy, then users who compile their own binaries from that source code can be confident those binaries are in turn trustworthy.

Most users of open source software do so by using precompiled binaries. Knowing that the source for those binaries was audited provides no guarantees about those binaries: it's possible that a malicious developer, package build infrastructure system administrator, external adversary, or compromised toolchain could introduce an undesired change in the binary package. With a reproducible build anyone can rebuild the binary from the audited source, ensuring that it indeed corresponds to that source.

Mike Perry and Seth Schoen's talk *Reproducible Builds: Moving Beyond Single Points of Failure for Software Distribution*[2] at the 31st Chaos Communication Congress (31C3) represents an inflection point in growing interest around reproducible builds from the perspective of software security and assurance. Perry and Schoen focused on the Tor Browser Bundle, and demonstrated the introduction of a vulnerability via a single-bit change in the binary. Research and reports have shown how malicious code introduced into developer environments can compromise source code during the build process, without leaving evidence in any on-disk file.

Malware may be introduced in compiled binaries via a compromised tool chain. A recent example is XCodeGhost, a modified version of Apple's proprietary XCode development environment, that inserts malware into compiled applications[3]. Although XCode is available for download from Apple at no cost, the installer is nearly 3GB in size and some developers chose to obtain a copy from a non-official local source. Many of these mirrors contained a version of XCode that had

been modified to automatically link a malicious payload.

Ken Thompson's seminal paper *Reflections on Trusting Trust*[4] demonstrated a compiler backdoor which, in brief, functions as follows:

1. The compiler is modified to insert a backdoor when compiling a specific, targeted application (for example, `login`).
2. The compiler is modified to determine when it is compiling itself, and insert the code to backdoor the targeted application.
3. The compiler is recompiled.
4. The code to insert the backdoors is removed.
5. The compiler is recompiled.

At the conclusion of this process the compiler will insert a backdoor when compiling the targeted application and when compiling itself, even though the source no longer exists for either backdoor.

David A. Wheeler's PhD dissertation[5] presents a solution to the Thompson attack, known as *Diverse Double-Compiling* (DDC). DDC requires that the (possibly backdoored) compiler under test builds reproducibly, and that the compiler can build itself. Verification proceeds as follows:

1. Build the compiler under test with itself.
2. Build the compiler under test with a trusted compiler.
3. Build the compiler under test with the result of step 2.

In the absence of compiler backdoors, the binary produced in step 1 and step 3 will be identical.

### 1.2 Operational Efficiency

In the FreeBSD base system the needs of the `freebsd-update` utility drove the initial effort on build reproducibility, rather than direct security and software assurance concerns. The `freebsd-update` tool is used to keep a FreeBSD installation up-to-date, with the primary use case being the application of fixes for security advisories (SAs) and errata notices (ENs). In operation, `freebsd-update` compares cryptographic hashes of installed files with those expected in the specified release, and fetches and installs those files that don't match the expected hash. In order for this mechanism to operate efficiently we require that binaries built from source files unchanged by the SA or EN are themselves unchanged, and thus not included in the update. In addition, binary differences in files that are changed will be minimized.

Reproducible builds can facilitate reductions in package mirror traffic and storage requirements. For example, a system may periodically build FreeBSD packages against the development Subversion ports and base system trees. Reproducible builds provide the same benefit here as for the base system, avoiding the creation of new package binaries when the subset of source files related to that package are unchanged.

In FreeBSD an *exp-run* is an experimental build of the entire ports tree, with some change applied. In the absense of a reproducible build, we can verify that ports still build with the change, and that some subset of ports pass their included test suites. If we have a ports tree that builds reproducibly, we can additionally determine the impact of changing toolchain components, header and macro changes, and identify packages using static libraries.

## 2 Reproducible Builds

Around 2014 Jérémy "Lunar" Bobbio and Holger Levsen, along with other members of the Debian project, began a holistic effort on *Reproducible Builds*, working towards a completely reproducible Debian release. They created the `https://www.reproducible-builds.org` web site to collect documentation and host a mailing list for the project. With sponsorship from the Linux Foundation's Core Infrastructure Initiative (CII) the Reproducible Builds project has been able to hire several developers to focus on reproducible builds on a full-time or part-time basis, and has hosted two summits to bring together developers from diverse projects with a common interest in reproducible builds.

Projects now working on reproducible builds include Debian, FreeBSD, NetBSD, OpenWRT, Fedora, Arch Linux, Coreboot, F-Droid, Bitcoin, Tor, Signal, OpenSUSE, Ubuntu, Guix, NixOS, ElectroBSD, Qubes, TAILS, Subgraph, and many others.

### 2.1 Components of a Reproducible Build

Reproducible builds require a deterministic build system, a reproducible build environment, and a way to distribute the build environment. Then, in order to provide value in software assurance some process is required for for regularly rebuilding software packages and checking the results.

A deterministic build system implies that inputs and outputs are stable. The input to the build (that is, the source code and other files) is known and unchanging. The output depends only on the input and a set of known, controlled external factors such as the version of the tool chain used to build the software.

### 2.2 Sources of Non-Reproducibility

There are many reasons software may not build reproducibly. Example sources of nonreproducibility include:

- Embedding build information into the binary (such as the current date and time)
- Input file ordering (filesystem or locale related)
- archive metadata
- unstable output ordering (for example, from hash-based data structures)
- intentional randomness introduced into a build
- DWARF debug info paths
- races / nondeterminism in threaded tools
- optimizations
- value initialization
- embedded signatures

## 3 Addressing Nonreproducibility

The reproducible builds team identified several common sources of nonreproducibility while iterating on reproducible build efforts. Template solutions for addressing these issues are presented here.

### 3.1 Stable order for inputs

Inputs to a build should be processed in the same order. Directory ordering is not necessarily stable, depending on the filesystem in use. For example, this command will not necessarily produce a reproducible output:

```
%tar -cf archive.tar src
```

This can be addressed by listing inputs explicitly:

```
%tar -cf archive.tar src/util.c
src/helper.c src/main.c
```

Or by explicitly sorting, with an explicit locale:

```
%find src -print0 LC_ALL=C sort -z
| tar -null -T - -no-recursion -cf
archive.tar
```

### 3.2 Deterministic version information

Version numbers should not be generated on each build. Instead, version information should be derived from the source - for example, a Version Control System revision number commit hash, a hash of the source code, or extracted from a changelog entry.

For example, FreeBSD's `newvers.sh` script obtains Subversion, Git, or Mercurial (Hg) version information and uses that as the build's unique identifier:

```
svn=`cd ${SYSDIR} && $svnversion 2>/dev/null`
...
#define VERSTR "...${svn}${git}${hg}..."
```

### 3.3 Eliminate build information

Many software packages encode build timestamps and other information for use in version or information strings. For example,

```
Compiled by emaste on 2 May 2016 at
14:12:03
```

Information such as the build date and time, user name, path, and hostname can simply be omitted. If the build is reproducible this information does not matter.

### 3.4 Don't record the current date and time

Timestamps should generally be avoided, but if one is required, or if upstream developers are not willing to entertain removing them, a suitable timestamp should be chosen and used in the build. This could be the date of the last commit in a version control system, extracted from a changelog, or simply hardcoded to an arbitrary

value. This timestamp can be passed to the build using the SOURCE_DATE_EPOCH environment variable.

The date and time is often recorded in archive files. This is commonly observed in static library .ar archives. Many archiving tools provide a command-line option to produce deterministic output, recording dummy values in place of timestamps:

```
ar -crD lib.a obj1.o obj2.o
```

If the archiver does not support such an option touch can be used to set a timestamp:

```
touch -d "2015-08-13 00:00Z" build/*
```

### 3.5 Explicitly set environment variables

Some environment variables can affect build output, for example:

- LC_CTIME (time strings)
- LC_CTYPE (text encoding)
- TZ (timezone)

Explicitly set these to a controlled value. However, upstream software projects should not override LANG; the user (or distribution software packager) should be able to build their software for a given language, and the resulting packages are intentionally not reproducible across different languages.

### 3.6 Stable order for outputs

Build tools may use hash tables to store items, and iterate over those items in some non-determistic order. One common case is found in scripting languages which intentionally introduce randomness into hashes, as a defense against algorithmic complexity attacks. This may be addressed by explicitly sorting the hash keys and iterating over that sorted list.

### 3.7 Avoid true randomness

Randomness may appear in a build from several sources:

- Temporary file names
- Generated UUIDs
- Filesystem images
- Protection against complexity attacks
- Link-Time Optimization (LTO) symbol names
- Unique stamps in coverage data files
- Compiler pseudorandom number generator (PRNG) use

These cases can be resolved by ensuring the random data does not appear in output artifacts (for example, by storing using only contents of temporary files, never their filenames), or by avoiding the use of random data altogether (for example, by deferring UUID generation).

Compilers may accept a seed value for PRNG initialization, which may be seeded with a fixed value, extracted from source code (a filename or content hash) or provided by make.

```
gcc -flto -frandom-seed=$seed
```

### 3.8 Avoid volatile inputs

Build inputs fetched from the network might change or disappear at any time. Either avoid relying on external resources (for instance, by committing a copy to the local source repository), or verify the content using a cryptographic checksum.

### 3.9 Controlled value initialization

Build tools may create binary structures or buffers in memory, and later write them to disk. Some tools have failed to fully initialize these, resulting in arbitrary data from the heap or stack written to the output file. Compiler-generated padding inserted between struct members is one potential source of such uninitialized data. All of these cases are simply bugs in these build tools that need to be fixed by fully initializing structures and buffers.

## 4 Tooling

The Debian-initiated Reproducible Builds project produced tools and specifications aimed at making builds reproducible, as well as tools for determining how non-reproducible builds differ, and tracking the progress toward a reproducibly-built package repository.

### 4.1 SOURCE_DATE_EPOCH

Build systems may make use of the notion of the current time (“*now*”). For example, the the C/C++ macros

__DATE__ and __TIME__ can embed a build timestamp into the binary.

The SOURCE_DATE_EPOCH specification[6] provides a standard way for build systems to provide a control timestamp, to be used instead of the current time. A UNIX timestamp (seconds since the epoch) is exported as an environment variable SOURCE_DATE_EPOCH, and build processes use it instead of the current time.

The specification claims that the value "SHOULD" be set to the last modification time of the source, incorporating any packaging-specific modifications.

### 4.2 disorderfs

disorderfs is a FUSE filesystem that introduces intentional nondeterminism into filesystem metadata (for example, returning directory entries in a random order). This can be used to identify cases where directory ordering unintentionally affects the build output.

### 4.3 strip-nondeterminism

For cases where it isn't feasible to modify a build tool to produce reproducible output the Debian project created strip-nondeterminism. It nomralizes files (gzipped files, ZIP archives, etc.), replacing nonreproducible fields with static values.

### 4.4 Diffoscope

For examining the differences between two expected-identical builds the Debian team built the Diffoscope tool. Diffoscope examines differences in depth, recursively unpacking archives and other container formats. It produces an HTML or plain text report containing human readable content – for example, it uncompresses and converts to text PDF files, and disassembles binaries.

Diffoscope is available in the FreeBSD ports tree as sysutils/diffoscope and can be installed as a binary package via `pkg install py35-diffoscope`. Debian also maintains an online diffoscope instance at https://try.diffoscope.org.

### 4.5 tests.reproducbile-builds.org

https://tests.reproducible-builds.org is a Jenkins Continuous Integration instance which builds packages or binaries for Debian, FreeBSD, NetBSD, and other projects. Each package is built twice, varying many aspects of the environment between builds, and the resulting packages are compared to determine how much of the project builds reproducibly.

Variations include the hostname and domainname, the TZ LANG LC_ALL and USER environment variables, the build timestamp (year, date, time), the uid and gid, the kernel version, 32- and 64-bit kernels, the shell, umask, CPU type, and filesystem.

## 5 Reproducible FreeBSD

Reproducible build efforts in FreeBSD started several years ago on a somewhat ad-hoc basis, but over the last two years build reproducibility has become a topic of increasing interest.

In FreeBSD we have two areas of focus for reproducibility efforts: the base system, and the ports collection. The FreeBSD base system is entirely under our control and we therefore have significant freedom to make changes.

The Ports collection consists of over 26,000 individual third-party software programs, requiring effort in the FreeBSD repository as well as coordination with upstream authors.

### 5.1 Base System

The FreeBSD *ReproducibleBuilds* wiki page was created in 2013, representing the beginning of a concerted effort addressing base system nonreproducibility. As of January 2017 all known reproducibility issues in the base system have been resolved. However, some changes are enabled only when the WITH_REPRODUCIBLE_BUILD compile-time setting is enabled.

Many utilities, configuration files and documentation were previously changed in order to build reproducibly for freebsd-update. Some examples of removing build information for reproducibility include:

- Build date in /usr/include/osreldate.h
- Build host and user in /usr/sbin/amd
- Build date and time in /usr/sbin/bhyve
- Build date and time in /etc/mail/*.cf
- Build date in /usr/share/doc/psd/13.rcs/paper.ascii.gz

- Build host, user, path and time in `/var/db/mergemaster.mtree`

By default the boot loaders and kernel contain some combination of the build user and hostname and the date and time. From the loader:

```
BTX loader 1.00  BTX version is 1.02
Consoles: internal video/keyboard
BIOS drive C: is disk0
BIOS 638kB/1046464KB available memory

FreeBSD/i386 bootstrap loader, Revision 1.1
(root@logan.cse.buffalo.edu
  Thu Jan  1 09:55:10 UTC 2009)
Loading /boot/defaults/loader.conf
```

And kernel:

```
% uname -v
FreeBSD 9.1-RELEASE #0 r243825:
    Tue Dec  4 09:23:10 UTC 2012
root at farrell.cse.buffalo.edu:
    /usr/obj/usr/src/sys/GENERIC
```

`/etc/src.conf` now supports a `WITH_REPRODUCIBLE_BUILD` compile-time option to disable this additional metadata, making the loader and kernel build reproducibly.

An interesting example of nonreproducibility was found in the `mandoc.db` manpage database, produced by `makewhatis`. When generating the database `makewhatis` used each page's inode number as a hash key, to detect hard linked pages and only index them once. It processed and output the pages ordered by hash key, resulting in non- deterministic output even on filesystems that return directory entries in sorted order.

The solution to this problem was suggested by Dag-Erling Smørgrav at the FreeBSD developers' summit during FOSDEM 2016:

1. Provide `fts_open()` with a comparison function to process directories and files in a deterministic order

2. In addition to the existing hash, insert pages into a linked list which will be sorted (by virtue of 1)

3. Iterate over pages by the list in 2, instead of hash order

This approach was implemented in the FreeBSD repository as r307003, and incorporated into the upstream mandoc repository.

### 5.2 Ports

The reproducible builds effort in the ports tree started somewhat later; the *PortsReproducibleBuilds* wiki page was created in March 2015. Steve Wills posted an initial patch (in FreeBSD code review D2032) that sets timestamps on files in the stage directory to that of the newest distfile. This addressed many sources of non-reproducibility in the package archive metadata (under the FreeBSD project's control), without addressing reproducibility of the archive content.

A test build that varied the hostname, time, and date was run at that time. Those results are found in the *D2032* column of Table 1. 64% of the packages built reproducibility. Only those ports that used a timestamp-preserving method for fetching distfiles had the possibly of reproducibly building packages with this approach.

A second iteration by Baptiste Daroussin recorded the timestamp in the `make makesum` command used during a port update, and used this to set the timestamps in the package archive file. We observed that of the ports that built, 79% built reproducibly (the *pkg* column in Table 1 and Figure 1).

The next attempt set SOURCE_DATE_EPOCH in the build environment, using the same port-update timestamp. This made use of any existing reproducible build support in individual ports, as well as the GNU Compiler Collection (GCC's) support of SOURCE_DATE_EPOCH for setting the __DATE__ and __TIME__ macros. This is the *+build env* column in Table 1. With this change 2499 additional ports built reproducibly, bringing the total to 80%. (Note that only those ports that previously built, but not reproducibly, were retested.)

A patch adding support for SOURCE_DATE_EPOCH to the Clang compiler was proposed in LLVM review D20791, and applied to the base system Clang. This increased the fraction of reproducible packages to 82%.

## 6 Next Steps

We have several tasks remaining in the FreeBSD reproducible builds effort. In the base system we need to enable reproducible build options by default in the release process, and ensure that continuous integration testing is in place to avoid regressions in reproducibility.

In ports we have a few more systemic changes to

make to allow a baseline of reproducibility. The work in progress patches used in the reproducibility investigation must be committed to the ports tree, and the tool chain patches need to be pushed upstream. This will allow us to reproducibly build over 80% of the package collection. Following Debian's example this will be followed by a long tail of addressing nonreproducibility in individual ports.

We need to develop reproducibility test rebuild infrastructure. This will rebuild the package collection and report results which can be compared with the expected or initial build.

Finally we need to introduce tooling to present reproducibility results to the end user. For example, the pkg install command could be configured to only allow the installation of packages that build reproducibly.

# Acknowledgments

FreeBSD developers Baptiste Daroussin, Colin Percival, Dag-Erling Smørgrav, and Steve Wills have contributed to base system and ports reproducibility efforts during my involvement in the project. Some of the examples of sources of nonreproducibility and corresponding techniques for addressing them are based on previous presentations by Holger Levsen. Funding for work on build reproducibility in FreeBSD has been provided in part by the FreeBSD Foundation and by the Linux Foundation as part of the Core Infrastructure Initiative.

# References

[1] `https://reproducible-builds.org/`

[2] Mike Perry, Seth Schoen, Hans Steiner, *Reproducible Builds: Moving Beyond Single Points of Failure for Software Distribution*, `https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner`

[3] `https://en.wikipedia.org/wiki/XcodeGhost`

[4] Thompson, Ken. (1984). *Reflections on Trusting Trust*, Communications of the ACM, 27(8):761–763.

[5] Wheeler, David A. (2009). *Fully Countering Trusting Trust through Diverse Double-Compiling* (Doctoral dissertation). `https://www.dwheeler.com/trusting-trust/dissertation/wheeler-trusting-trust-ddc.pdf`

[6] Lamb, Chris et al. SOURCE_DATE_EPOCH specification. `https://reproducible-builds.org/specs/source-date-epoch/`

Table 1: Ports reproducibility progress

| | D2032 | Stock | SOURCE_DATE_EPOCH pkg | +build env | +Clang |
|---|---|---|---|---|---|
| Queued | | 26046 | 26046 | 8600 | 8565 |
| Non-reproducible | 15164 | 25222 | 5162 | 3534 | 4011 |
| Reproducible | 8435 | 0 | 20009 | 5062 | 4549 |
| Failed | | 824 | 875 | 4 | 5 |
| Newly reproducible | | | 20009 | 116 | 514 |
| Total reproducible | | | 20009 | 20125 | 20639 |
| Packages built | 23599 | 25222 | 25171 | 25167 | 25166 |
| Reproducible % | 64.3% | 0% | 79.5% | 80.0% | 82.0% |

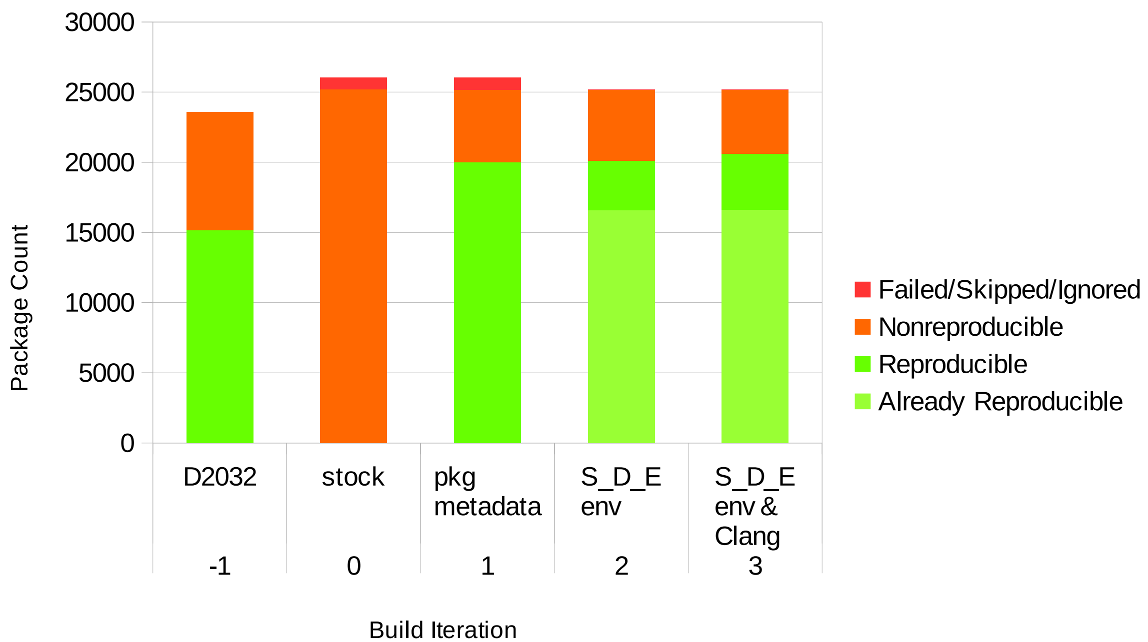


Figure 1: Ports reproducibility progress