Hi! I’m Lunar. I’m very much alive and feel really honored to be here.

Couple of words about me: I believe that humans should be controlling machines and not the other way around. That’s why I’m a strong supporter of free software. I’m officially a Debian Developer since 2007, and involved in the Tor Project since 2009. I need to say the work I’m presenting is also the work of many different people involved in Debian and in other projects.

And it’s my involvement in Tor that got me interested in the topic I am going to talk about today. So what is the issue that we faced there?

The problem

When we talk about software, there are two sides of it. Source is what some humans can read. Binary can also be read, but only by a really tiny fraction of humanity. Binary form is what the computers need to run the software. Transforming the source code into binary format is code “compiling” or “building”.

The great thing with free software is that we have the freedom to study that the source code is doing what it is supposed to be doing. That it does not contain any malware, malicious code, or security bugs. Free software also gives us the freedom to run the software in any way we want.

So, we have the source code that we can verify and we have a binary we can use. Question: when we download software in binary form, how do we know how it was built? Well, right now in almost every cases, your only choice is to trust the software author, or the distribution, that the archive with the source that has approximately the same name is what has been used to create the binary.

Wouldn’t it be much better if we could get a proof?

Some people could say that we need to trust the soft-
ware author, or the distribution, in all cases. Well, we indeed have to trust the software author and the distribution channel. But we have processes to do that, cryptographic signatures and all that jazz. But as Mike Perry and Seth Schoen explained in greater length during a talk last December at the 31C3, developers might be targeted and not realize that their building environment has actually been compromised. During the talk, Seth showed a proof-of-concept kernel exploit that would modify—without touching anything on the disk—a source file while it was being read by the compiler.

Just one example
At a CIA conference in 2012:
firstlook.org/theintercept/2015/03/10/ispy-cia-campaign-steal-apples-secrets/
Lunar (Debian) Reproducible builds HOWTO CCCamp15 5 / 59

And we are not discussing hypothetical attacks here! A couple of months after Mike & Seth’s talk, The Intercept released another document from the Snowden leaks describing the program of an internal CIA conference in 2012. The presentation that we see here was about “Strawhorse” and describes an attack on XCode—the software development environment for Mac OS X and iOS. They had a modified version, ready to be implanted on developer’s systems, that would create binaries being watermarked, or leaking data, or containing trojans... And this is all without the developer realizing that this is happening. So even if we trust our developers: they might totally be of good faith... and we would still totally get owned.

The solution
We call this:
“reproducible builds”

It’s trendy!
• Bitcoin (done)
• Tor (done)
• Debian (in progress)
• FreeBSD (in progress)
• NetBSD (in progress)
• Coreboot (done)
• OpenWrt (in progress)

Good news: it’s getting trendy. I became familiar with the concept because of the work done by Mike Perry to get Tor Browser to build reproducibly. Himself, he was inspired by concerns in the Bitcoin community. It’s been two years that we’ve started to work on this in Debian. Some people have started to work on FreeBSD. Coreboot fixed all the reproducibility problems in the past months. OpenWrt has started to accept some patches to make this possible. And it’s not limited to these projects. We are seeing many people who are interested in making their project “reproducible” or making their tools able to produce identical binaries.

It should become the norm.

So what can we do about it? We need be able to get reasonable confidence that a given binary was indeed produced using its supposed source. To achieve this, we want to enable anyone to reproduce identical binary packages from a given source. If we have this, and then enough people to perform another build on different computers, on different networks, at different times, then we can assume that either everybody is compromised the same, or—with better luck—that no bad stuff got added behind our backs.

And that’s a very good thing because “reproducible builds” should become the norm. One say the only software that can be secure are free software because we can perform proper audit. But this really only apply when we can trust the binaries. As software developers, we do want to provide a verifiable path from the source to the binaries we distribute.
While working on this for the past two years in Debian, and kind of becoming a reference on the topic without realizing it, we identified that there were multiple aspects to getting “reproducible builds”. First you need to get the build to output the same bytes for a given version. But others also must to be able to set up a close enough build environment with similar enough software to perform the build. And for them to set it up, this environment needs to be specified somehow. Finally, you need to think about how reboots are performed and how the results are checked. I’m not going to talk about this last point in this presentation. It’s mostly a matter of documentation because we advocate that checking the results should be as easy as comparing if the build products are byte for byte identical.

So first, how do we get a build system to always build the same thing.

Deterministic build system

In a nutshell:
- Stable inputs
- Stable outputs
- Capture as little as possible from the environment

In a nutshell, you need to make sure the inputs are always the same. That the outputs are always the same. And that as little as possible from the environment is being captured. Sounds like common sense?

Yet, with the work we’ve done in Debian, we’ve seen that these assumptions do not hold for a lot of the software we build. The number one issue preventing the output to always be the same is “timestamps”. The date and time of the build creeps everywhere, we’ll get back to this. Other common problems are variations in file ordering on disk, usage of randomness, specialized code for a given CPU class, the directory in which the build is being performed getting embedded in binaries, or other settings like locale or timezone affecting what gets recorded. But first, before giving some solutions to all these problems...

To build some piece of software, we actually need to get our hands on its source... WhyTheLuckyStiff was an amazing member of the Ruby community. If you meet someone who doesn’t like Ruby, that’s because they never had the chance to read _why’s Poignant Guide to Ruby. Why am I talking about _why? Because one day, he disappeared and took all his websites, writings and code down...

Volatile inputs can disappear

- Don’t rely on the network
- If you do:
  - Verify content using checksums
  - Have a backup
- The binary distributor should provide a fallback

Inputs from the network—even if it doesn’t seem like it—are volatile. So don’t make your build system rely on remote data. Or if you do, use checksums to make sure the content has not been modified and keep backups. Ideally, provide a fallback location with these backups. A good example is how the FreeBSD ports work: they record MASTER_SITES for a given piece of software, the size and a cryptographic checksum for each files downloaded from...
these master sites, but they also keep a copy of each files on their mirrors. That’s the best way to handle network inputs, but if you can avoid them, do it. Ok, now let’s tackle some of the common issues...

Here we can see the differences between two Tar archives. They both contain exactly the same files. But, as you can see, not in the same order.

Stable order for inputs
- Always process multiple inputs in the same order
- Directory listings are not stable!

Example
```bash
tar -cf archive.tar src
```

This is an example of having a different output because the order of inputs is not stable. When doing the basic operation of listing a directory, there is no guarantees on the order in which they will be returned. So if you use `tar` as shown at the bottom, you don’t know in which order files in the `/src` directory will be written in the archive.

Stable order for inputs
- Always process multiple inputs in the same order
- Directory listings are not stable!

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

One solution to this is to list all inputs explicitly. The construction here is actually pretty common for source code already.

Stable order for inputs
- Always process multiple inputs in the same order
- Directory listings are not stable!

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

Another option is to use sorting. If you want to do it right for `tar` you actually need to use `find`, `sort` and `tar` in succession like shown. But there’s a catch!

Stable order for inputs
- Always process multiple inputs in the same order
- Directory listings are not stable!
- Solutions:
  - List inputs explicitly
  - Use sorting
  - But watch out for difference between locales.

Example
```bash
find src -print0 | sort -z | tar --null -T - --no-recursion -cf archive.tar
```

Depending on the locale, the `sort` command will sort files differently. Typically, some locales will sort all uppercase letters together, while some other will be case-insensitive. So don’t forget to specify the locale when using `sort`.

Stable order for inputs
- Always process multiple inputs in the same order
- Directory listings are not stable!
- Solutions:
  - List inputs explicitly
  - Use sorting
  - But watch out for difference between locales.

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

Here’s another example taken from Coreboot and it’s the kind of issue you really don’t want to have to track down. Mike Perry and Georg Koppen faced such an issue with the Windows build of Tor Browser. The difference we’re seeing here is only a couple of bytes. And these bytes will be different with almost all builds, and with no predictable or common patterns. That’s because they are actually the content of whatever contains the memory at that time.
And using random values from memory will not produce deterministic output. So don’t record memory by accident. Here we can see Coreboot code that was actually producing the dump we’ve just seen.

And as you can see the fix is trivial. So remember to always initialize all data structures you are using. Because tracking down the source of these kind of problems can really be a pain.

Another example. Here we see a build number embedded in a German dictionary for aspell, and that gets to be different from one build to another.

Don’t do that. We want stable output, so it’s a bad idea to create a new version or “build number” on each build.

Instead, be deterministic and extract an information actually meaningful to the source that is being built. It can be the revision number from version control system. A hash of the source code might even be a better idea. Good thing about Git: they are the same. Another option is to extract stuff from a “changelog”. The example here is an extract from how it’s done for the nsis Debian package.

That’s a dump of the nsis binary. The difference between these two builds is fairly obvious: on the left, we have July 29, and on the right, it’s using the date of the next day, July 30.

Don’t record the current date and time

Avoid timestamps

It’s one of the many many examples where the date and time of the build is being recorded by the build process, leading to different outputs. So, to sum it up: timestamps, bad idea. The current date and time is not really a useful piece of information anyway: you can always take an old piece of software and build it today. If the date and time of the build is meant to be an indication of the environment in which the build was made, then, as you’ll see, more precise ways are needed anyway to get reproducible builds.
Don't record the current date and time

- Avoid timestamps
- If you need one:
  - Use date of last commit in VCS
  - Extract from changelog

So, if you really need to have a date and time recorded, then like for version numbers, make the date relevant to the version control system. Or extract it from a changelog.

Don't record the current date and time

- Avoid timestamps
- If you need one:
  - Use date of last commit in VCS
  - Extract from changelog
  - Don't forget the timezone

But in that case, don’t forget to record and use the original timezone or do everything in UTC. Otherwise, depending on where the build is made, you are likely to get different results.

One tool to avoid timestamp-related issue is faketime. faketime is a library that is loaded through the LD_PRELOAD environment variable and that will catch calls asking the system for the current time of day, and reply instead a predefined date and time. In some cases, it works just fine and can solve problems without requiring many changes to a given build system. The problem is that some tools rely on accurate times. The very common Make being one of them. Make requires accurate times because it will do it’s best to only recompile stuff when the sources have changed since the last time a build happened. It gets really bad when doing parallel builds. The bug linked here is a reproducibility issue affecting the Tor Browser where faketime has the side effects that some objects are actually built multiple times because Make can’t properly determine if they are too old or not, and in the end the ordering of some object files differ. So I would recommend to avoid faketime as much as possible, but handled with care, for limited uses, it can be an option. So, what should we do instead when some tool we use does record the current time?

Don't record the current date and time

- Avoid timestamps
- If you need one:
  - Use date of last commit in VCS
  - Extract from changelog
  - Don't forget the timezone
  - faketime is an option but has serious drawbacks
    https://bugs.torproject.org/12240
  - Implement SOURCE_DATE_EPOCH

A much better idea is to implement or support SOURCE_DATE_EPOCH.

SOURCE_DATE_EPOCH

- What is it?
  - Environment variable with a reference time
  - Number of seconds since the epoch (1970-01-01 00:00:00 +0000 UTC)
  - If set, replace “current time of day”
  - Implemented by hw2man, Epydoc, Doxygen, Ghostscript (in Debian)
  - Patches ready for GCC, txt2man, libxslt, Gettext

https://wiki.debian.org/ReproducibleBuilds/TimestampsProposal

One tool to avoid timestamp-related issue is faketime. faketime is a library that is loaded through the LD_PRELOAD environment variable and that will catch calls asking the system for the current time of day, and reply instead a predefined date and time. In some cases, it works just fine and can solve problems without requiring many changes to a given build system. The problem is that some tools rely on accurate times. The very common Make being one of them. Make requires accurate times because it will do it’s best to only recompile stuff when the sources have changed since the last time a build happened. It gets really bad when doing parallel builds. The bug linked here is a reproducibility issue affecting the Tor Browser where faketime has the side effects that some objects are actually built multiple times because Make can’t properly determine if they are too old or not, and in the end the ordering of some object files differ. So I would recommend to avoid faketime as much as possible, but handled with care, for limited uses, it can be an option. So, what should we do instead when some tool we use does record the current time?

Don't record the current date and time

- Avoid timestamps
- If you need one:
  - Use date of last commit in VCS
  - Extract from changelog
  - Don't forget the timezone

A much better idea is to implement or support SOURCE_DATE_EPOCH.

SOURCE_DATE_EPOCH

- What is it?
  - Environment variable with a reference time
  - Number of seconds since the epoch (1970-01-01 00:00:00 +0000 UTC)
  - If set, replace “current time of day”
  - Implemented by hw2man, Epydoc, Doxygen, Ghostscript (in Debian)
  - Patches ready for GCC, txt2man, libxslt, Gettext

https://wiki.debian.org/ReproducibleBuilds/TimestampsProposal

So an easy fix for timestamps is to set SOURCE_DATE_EPOCH in your build system.
And if it doesn’t have the desired effects, please write and submit patches!

But, I’m sorry to say I’m not over with timestamps. Here you can see how the time of the build got recorded by gzip in its headers, and by Tar, as the time of the files in the archive.

For most archive formats will keep the file modification times in their metadata. For some rare tools, you can simply tell them to not record metadata, like gzip with its -n option. But for all others, that means that if your build system creates a new file, and then stores it in an archive, the current time will be recorded in the archive, as we just saw. Several solutions are possible but it also depends on the type of archive we are dealing with.

For Tar, we can just use a reference time for all members of the archive.

For some archive formats, there is always the option of doing post-processing. I will come back to talk about the strip-nondeterminism tool later.

Yet another issue. Can you understand what is happening here? There are three functions in this executable. It’s always the same three. But sadly, they are not always written in the same order, and the result is not byte-for-byte identical.

So beware of the order of outputs as it needs to be always the same. Everytime there’s a list, there might be ordering problems. The typical issue is related to key
ordering in hash tables. To prevent an attacker from consuming a large amount of CPU or memory by attacking the hash function used to store data in a dictionary-like structure (they are called hashes in Perl and in Ruby, or dicts in Python), the function is made slightly different on each run by using a random seed. That means that when retrieving the keys in the dictionary, they are likely to be in a different order on every run. The solution to this problem is pretty simple.

### Stable order for outputs

- Always output lists in the same order
- Typical issue: key order with hash tables
- Sort!

```
for module in sorted(dependencies.keys()):
    version = dependencies[module]
    print('%s (>= %s)' % (module, version))
```

Sort! It’s often just a single extra function call that will make the output deterministic.

### Avoid (true) randomness

- Randomness is not deterministic
- Seed your PRNG from known value
- Use a fixed value
- Extract from source code (filename, content hash)

```
$ gcc -f lto -c -random-seed=utils.o utils.c
$ nm -a utils.o | grep inline
0000000000000000 n .gnu.lto_.inline.381a277a0b6d2a35
```

Sometimes you still need to prevent collisions, and so it might be better to seed the value with a value that can change from one file to another, or from one version of the software to the next. Deriving a filename or a content hash is an obvious answer for these cases.

### Define environment variable affecting outputs

- Some environment variables will affect software outputs. E.g:
  - LC_CTIME for time strings
  - LC_CTYPE for text encoding
  - TZ for times

```
$ gcc -f lto -c -random-seed=utils.o utils.c
$ nm -a utils.o | grep inline
0000000000000000 n .gnu.lto_.inline.a108e942
```

Another thing you might want to look for is how environment variables might affect outputs. The `date` command on Unix systems might return different results depending on the locale. Files might be written using different character encodings. Various piece of code—hint Gettext—will be affected by the current timezone.

### Define environment variable affecting outputs

- Some environment variables will affect software outputs. E.g:
  - LC_CTIME for time strings
  - LC_CTYPE for text encoding
  - TZ for times

```
$ setenv LC_CTIME Europe/London
$ gcc -f lto -c -random-seed=utils.o utils.c
$ nm -a utils.o | grep inline
0000000000000000 n .gnu.lto_.inline.381a277a0b6d2a35
```

If you identify that you are using one of these tools, then set these variables to avoid surprises. One pledge though...

### Define environment variable affecting outputs

- Some environment variables will affect software outputs. E.g:
  - LC_CTIME for time strings
  - LC_CTYPE for text encoding
  - TZ for times

```
$ export LC_CTIME="Europe/London"
$ gcc -f lto -c -random-seed=utils.o utils.c
$ nm -a utils.o | grep inline
0000000000000000 n .gnu.lto_.inline.a108e942
```

Please don’t blindly overwrite the system language if
you can avoid it. I believe that people should be able to interact with computers in the language they prefer, and they might prefer to get compiler errors in their first language.

### Stop recording build system information

- Don’t record information about the build system, like:
  - date and time of the build
  - hostname
  - path
  - network configuration
  - CPU
  - environment variables
  - …
- If you really want to record them, do it outside the binaries

As a general recommendation, try not to record any information about the build and the build system. Date and time as we already said, but it also goes for the system hostname or its CPU.

If you really want to record them, it’s best to do it outside of the binaries that will be distributed to users so actual code can be compared more easily. A build log is a perfectly reasonable location to record these kind of information. The version string, not so much. Knowing in which environment a software has been built is far less interesting when you have “reproducible builds”.

### What’s in a build environment?

- At least: build tools and their specific versions

So what do we call a build environment? Well, at the very least, it’s the tools that are needed to build the software. And in most cases, the actual version that has been used. Compilers for example are being improved all the time with new optimizations. A piece of software built with a newer version of a compiler is quite likely be faster than one built with an older version of the same compiler. That’s a good thing, but that means different versions of the same compiler is likely to produce different binaries.

### What’s in a build environment?

- At least: build tools and their specific versions
- Up to you, depending on the build system:
  - build architecture
  - operating system
  - build path
  - build date and time
  - …

And then, you can decide that other aspects of the environment should be reproduced by users if they want to build your software. If you don’t support cross-compiling, mandating a given build architecture is probably a sane thing to do. Or declaring that a given binary can only be created by using FreeBSD. One thing we currently decided for Debian to avoid some pain is to mandate a particular directory where the build should be performed. This avoids problems with paths being recorded in debug symbols, for which we don’t have good post-processing tools at the moment. If you use things like `SOURCE_DATE_EPOCH` or `faketime`, you can also declare that a build must be performed using a definite reference time. And basically, it’s quite up to you, but you must identify what’s in there so users have a chance to produce the same output as the one you are distributing. Once identified you must give a way to reproduce the same environment on their own system.

### Build from source

- Build tools affecting the output from source
- Record version / tag / git commit
- Approach used by Coreboot, OpenWrt, Tor Browser

That’s because we are going to use a well-defined environment to perform our builds, as we want users to be able to reproduce it... so they can actually reproduce the build.
So one way to have users reproduce the tools used to perform the build is simply to have them start by building the right version of these tools from source. That’s the approach used by Coreboot, OpenWrt and partially Tor Browser.

**Reference distribution**

- Use a stable distribution (e.g. Debian, CentOS)
- Record package version
- Hope the old package will stay available / record
- Approach used by Bitcoin, **Tor Browser**

Another approach, used by Bitcoin and other parts of the Tor Browser build process, is to use a specific version of an integrated operating system. Usually with GNU/Linux using a stable distribution like Debian or CentOS. It needs to stay available for long and to have the least amount of update possible. Better record exact package version, and hope these versions can be later reinstalled.

**Virtual machines / containers**

- Using a VM saves some problems:
  - Same user
  - Same hostname
  - Same network configuration
  - Same CPU
- Introduce new things that need to be trusted

Some things can be quite simplified by using virtual machines or containers. With a virtual machine you can easily perform the build in a more controlled environment. Always using the same user, the same hostname, the same network configuration, you name it. The downside is that it can introduce a lot of software that has to be trusted somehow. For example, it’s currently not possible to install Debian in a reproducible manner. This makes it harder to compare different installations. We’ve done some preliminary work, but it’s only been about identifying issues so far.

**Proprietary operating systems**

- Cross-compiling to the rescue!
- For Windows:
  - mingw-w64: build Windows binaries on *nix
  - NSIS (Nullsoft Scriptable Install System)
- For Mac OS X:
  - Used to be quite complicated
  - Now pretty straightforward with crosstool-ng
  - Need a non-redistributable SDK extracted from XCode
  - .dmg are a bit tricky

And speaking about trusting operating systems, how can we handle the proprietary ones? It’s hard to assess they have not been tampered with. So let’s just avoid that path. We actually have free software tools that can build perfectly fine software for Windows and Mac OS X. This is already how it’s done for Bitcoin and Tor Browser, so thanks to them for researching this hairy topic. For Windows, **ming-v64** and the Nullsoft Scriptable Install System are both available in Debian. This is actually how we are building the application that can launch the **debian-installer** on Windows. For Mac OS X, it used to be quite hackish, but it’s getting better thanks to the work done by Ray Donnelly. You will need to use a non-redistributable part, although it’s provided by Apple after free registration, from XCode to build the toolchain. Software from Mac OS X is often distributed as disk images which can be created under GNU/Linux, but it kinda requires 3 different tools at the moment. Hopefully if more people start doing this, the whole process is likely to get improved in the future.

**Good ol’Makefile**

- Download known toolchain archives
- Compare reference checksums
- Build and setup
- Coreboot: make crossgcc

If the environment is only about build tools, maybe the easiest way is just to add an extra target to your **Makefile**. For example, building Coreboot almost mandate to first run **make crossgcc**. This will download known archives for GCC, **binutils** and others, compare the archives with reference checksums, and proceed to build them. And it’s these tools that are going to be used by the rest of the build process.
Check-in everything

- Check-in all the toolchain source code in VCS
- Approach used for the base system in *BSD, and Google
- Make sure everything is checked in (use sandbox on Linux)
- Recently open-sourced: Bazel
  - http://bazel.io/
- Can be hard to ask everyone to download everything all the time

A more radical extension to the former approach is to actually check everything in your version control system. Everything as in the source of every single tool. That’s how it’s working when you are “building the world” on BSD-like systems. That’s also how Google is doing it internally. To make absolutely sure that everything is checked-in, you can even use “sandboxing” mechanisms to avoid the risk of running a tool that has not been built from source. Google recently started open-sourcing the tool they use internally to drive such large scale builds under the name Bazel. So despite its syntax that I personally find hard to read, it’s probably worth checking out. But if you are not working in a corporate environment, or on a fully integrated operating system, it might be hard to push. You don’t really want to ask everyone to download and build every possible version of GCC every time they would like to build a piece of code.

Ship the toolchain as a build product

- Make the toolchain as a build product
- OpenWrt:

Example

```bash
$ wget https://downloads.openwrt.org/...tar.gz
$ make package/xz/compile
```

As a middle ground, OpenWrt offers an “SDK” that can be downloaded alongside their system images which contains everything that is needed to build—or rebuild—extra packages. To close the loop, as the SDK becomes another build product, it has to be possible to build it reproducibly.

Gitian

- Used by Bitcoin, Tor Browser
- Drives LXC or KVM
- “Descriptors” describing the build using:
  - Base distribution
  - Packages
  - Git remotes
  - Other input files
  - Build script

Resources

https://github.com/tianon/gosu/blob/master/Dockerfile
https://www.vagrantup.com/
https://gitian.org/

Gitian is the tool used by Bitcoin and the Tor Browser. It either drives a Linux container using LXC, or a virtual machine using KVM. Gitian takes “descriptors” as input which tells which base GNU/Linux distribution to use, which packages to install, which Git remotes must be fetched, any other input files, and a build script to be run with all of that. As explained earlier, using a virtual machine helps to get rid of several extra variations that can happen from one system to the next. But this is more complicated to setup for users.

Docker

- Provide a way to describe specialized Linux container images
- Build in a controlled environment
- Docker images can be addressed with a hash of their content
- Bazel has support to build Docker images reproducibly

Making containers easy to setup and use is exactly the problem that Docker is trying to solve. **Dockerfiles** are used to describe how to create a container, and how applications can be run in there. This specific example is how a tool often used in Docker images, gosu is built. Using the reference container made available to build Go applications, it then installs the necessary dependencies, and calls the Go compiler in that environment which should be pretty much the same all the time. To be sure that the base compiler is the same, one could use the fact that Docker images can actually be addressed by a hash of their content. Another option is to build the Docker image itself in a reproducible manner, and from what I read in the documentation, Bazel—mentioned earlier—is able to do this.

Vagrant

- Drive VirtualBox using Ruby and other scripts
- Build in a controlled environment
- Also works under OS X and Windows

Vagrant is another tool, written in Ruby, that can drive virtual machines with VirtualBox. It can also be used to get a controlled build environment. The upside of Vagrant and VirtualBox is that they work on Mac OS X and Windows, and so this might help more users to actually check that a build has not been tempered with.
Debian .buildinfo

- Tie in the same file:
  - Sources
  - Generated binaries
- Packages used to build (with specific version)
- Can be later processed to reinstall environment
- All versions are available from snapshot.debian.org

For Debian, we decided for another path. We defined a new control format, called .buildinfo where we list the sources, the generated binaries, and every packages that were installed in the system when these binaries were built. It means that we can easily reproduce the build environment by reinstalling each package with the specific version that we had recorded. We can do that because Debian has a service called “snapshot” which records every binary package ever uploaded to the Debian archive, so older versions of a given package stay available even when they have been superseded by a later one.

Example .buildinfo
Format: 1.9
Build-Architecture: amd64
Source: txtorcon
Binary: python-txtorcon
Architecture: all
Version: 0.11.0-1
Build-Path: /usr/src/debian/txtorcon-0.11.0-1
Checksums-Sha256:
a2654069.7b 125910 python-txtorcon_0.11.0-1_all.deb
28f6bca.69 2039 txtorcon_0.11.0-1.dsc
Build-Environment:
base-files (= 8),
baseseed (= 3.5.37),
bash (= 4.3-11+b1),
hostname

Here’s an example of what a .buildinfo looks like. So you can see the build architecture, checksums of source—that’s the .dsc—and the binary packages, the build path, and all the packages involved. Hopefully they will be soon available on Debian mirrors and users should then be able to simply call a script to re-do the environment and then the build.

Testing for variations

- Build a first time
- Save the result
- Perform change(s) to the environment
- Build a second time
- Compare results

If users are the ones that detects that changes in the environment affect the build, it’s going to raise a lot of false alarms. So better perform tests ahead. The basic idea we are currently using in Debian is that we build a first time, keep the result aside, change various stuff in the environment, perform a second build, and then compare the results.

reproducible.debian.net

- Continuous test system driven by Jenkins
- Bad-ass hardware sponsored by ProfitBricks
- Tests about 1300 Debian source packages per day on average
- Results are visible on a website
- Other projects: Coreboot, OpenWrt, yours?

Based on this, Holger Levensen, now helped by Mattia Rizzolo, setup a continuous test system driven by Jenkins. Thanks to ProfitBricks for the crazy bad ass hardware as it’s able to perform 1300 tests—that means building 1300 packages twice—every day on average. The results are then put in a database and browsable on the web. The system has been recently extended to other projects and we are currently performing tests for Coreboot and OpenWrt. Work has also started to test FreeBSD and NetBSD… For now, we’re testing Linux but…

To give you an overview, these are all the variations we are currently performing between the first build and the second one. So hostname, domain name, timezone (see how we use timezones that are more than 24 hours apart so that we get a different day), the language, general locale settings, username, user id, group id, network namespace, kernel version, umask… and pretty soon we’ll have the second run on a different machine that will have

It’s been two years we’ve been working on this in Debian. We’re not totally there yet, but here’s still some tips to share.

Variations on reproducible.debian.net

<table>
<thead>
<tr>
<th>variation</th>
<th>first build</th>
<th>second build</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainnames</td>
<td>debian.net</td>
<td>i-capture-the-domainname</td>
</tr>
<tr>
<td>env LANG</td>
<td>en, GB.UTF-8</td>
<td>fr, FR.UTF-8</td>
</tr>
<tr>
<td>env LC_ALL</td>
<td>en</td>
<td>fr, FR.UTF-8</td>
</tr>
<tr>
<td>env USER</td>
<td>root</td>
<td>root</td>
</tr>
<tr>
<td>gid</td>
<td>1111</td>
<td>2222</td>
</tr>
<tr>
<td>UTS namespace</td>
<td>shared with the host</td>
<td>modified using /usr/bin/unshare --uts=uts</td>
</tr>
<tr>
<td>kernel version</td>
<td>Linux 3.16.0-4-amd64</td>
<td>Linux 3.16.0-5-amd64</td>
</tr>
<tr>
<td>unmask</td>
<td>0022</td>
<td>0022</td>
</tr>
<tr>
<td>CPU type</td>
<td>same for both builds (work in progress)</td>
<td>same for both builds (work in progress)</td>
</tr>
<tr>
<td>year, month, date</td>
<td>2015-11-29</td>
<td>2015-11-29</td>
</tr>
<tr>
<td>hour, minute</td>
<td>00:02</td>
<td>00:02</td>
</tr>
</tbody>
</table>
| everything else | hourly the same.. usually, the minute differs.. (work in progress) | usually the same..
a different number of cores, a different date, and maybe a different filesystem. Hopefully, we'll then be covering it all, but time will tell.

I didn’t want to make this talk too much about the Debian project, but just to give you a quick view on how we are doing. Some crucial patches are still experimental and not in the main Debian archive. But with our experimental toolchain, our tests are now positive for more than 800. Contributions from all projects are highly welcome.

And here might be a more accurate view of the progress we are making every day as Debian maintainers integrate the patches that we are submitting.

**Debugging problems: diffoscope**

- Examines differences in depth
- Outputs HTML or plain text showing the differences
- Recursively unpacks archives
- Seeks human readability:
  - uncompresses PDF
  - disassembles binaries
  - unpacks Gettext files
  - easy to extend to new file formats
- Falls back to binary comparison

http://diffoscope.org/
(formerly known as debootstrap)

That’s also because we came up with a tool that helped us understand issues. Comparing two different compressed archives is not going to help you understand much of why they are different. It’s every files in these archives that need to be compared. So that’s why we came up with “diffoscope”. It will recursively unpack archives, and for binary files, try to get a human readable representation before comparing them. It will fallback to comparing hex-dumps if the bytes are different but no differences show up in the human readable representation. “diffoscope” is designed to be extensible and has now grown way beyond just being a tool for Debian packages. It also supports RPM, ISO images, or squashfs filesystems.

Another tool we came up in Debian is **strip-nondeterminism**, originally written by Andrew Ayer. It is meant to be a central place to perform post-processing on various file formats to remove bits of non-determinism. It already handles several file formats and should be easily extensible to more. It is written in Perl, like the other tools you need to create Debian packages, to avoid adding an extra build dependency.

**Resources**

- Reproducible Builds HOWTO (work in progress)
  https://reproducible.debian.net/hwto/

I’m ending with a couple more resources. We have started to write an HOWTO which should contain more or less what I’ve been telling you for the past forty minutes. Contributions from all projects are highly welcome.
As I said, “reproducible builds” should become the norm, and it would be great to have reference documentation that can be widely shared and used.

We’ve been collecting a lot of information about reproducibility issues on the Debian wiki. Some are pretty specific to Debian, but there’s a lot to learn. We’ve also been collecting rationales, media mentions, and other goodies.

Last but not least, I’d like to mention David A. Wheeler’s work on Diverse Double-Compilation. Often when I explain the idea of “reproducible builds”, someone comes up asking “but how can you be sure that your compiler has not been backdoored so that the next time it builds a compiler it will not insert another backdoor?” This is also known as the “trusting trust” attack from Ken Thompson that was mentioned in the Snowden document. So David refined (and also did a formal proof) that we can answer this question using a process that is called Diverse Double-Compilation. I’ll try to sum it up quickly: you need two compilers, with one that you somehow trust; then you build the compiler under test twice, once with each compiler, and then you use the compilers that you just built to build the compiler under test again. If the output is the same, then no backdoors. But for this scheme to work, you need to be able to compare that both build outputs are the same. And that’s exactly what we are enabling when having reproducible builds.

I’m done. I sincerely hope that this short lecture will make you want to provide reproducible builds. As I said, I really think we all need this to become the norm. Thanks for listening. And I now will be happy to take questions.

Thanks!

- Debian “Reproducible Builds” team
  (you are just so awesome!)
- Mike Perry, Georg Koppen, David A. Wheeler
- Linux Foundation and the Core Infrastructure Initiative

lunar@debian.org 0603 CCFD 9186 5C17 E88D
4C79 8382 C95C 2902 3DF9
clothes: Elhonna Sombrefeuille — hair: igor