Increasing the Integrity of the Open Source Software Supply Chain with Reproducible Builds

Journées Nationales GDR Sécurité Informatique
2022-06-22

Stefano Zacchiroli — Télécom Paris, Institut Polytechnique de Paris
stefano.zacchiroli@telecom-paris.fr
https://upsilon.cc/zack
@zacchiro | mastodon.xyz/@zacchiro

Open Souce Software Supply Chain Attacks

- [Ohm20]: Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier. *Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks*. DIMVA 2020: 23-43.
- [Ladisa22]: Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais: *Taxonomy of Attacks on Open-Source Software Supply Chains*. (Preprint, to appear.)

The software supply chain

Supply chain attacks

A **software supply chain attack** is a particular kind of *cyber-attack* that aims at *injecting malicious code* into an otherwise *legitimate software product*.

Notable examples

- *NotPetya* (2017): ransomware concelaed in an update of a popular accounting software, hitting Ukranian banks and major corps (B$)
- *CCleaner* (2017): malicious version of a popular MS Windows maintenance tool, distributed via the vendor website
- *SolarWinds* (2020): malicious update of the SolarWinds Orion monitoring software, shipping a delayed-activation trojan. Breached into several US Gov. branches as well as Microsoft

Open source supply chain attacks

- Is this specific to Free/Open Source Software (FOSS)? No.
- But modern **FOSS package ecosystems** are heavily intertwined.
    - Examples: NPM (JavaScript), PyPI (Python), Crates (Rust), Gems (Ruby), etc.
    - 100/10k/1M packages, depending on each other due to code reuse opportunities.
    - **Reverse transitive dependencies** grow fast. A single package could be required by *thousands* of others.

`left-pad` (2016)

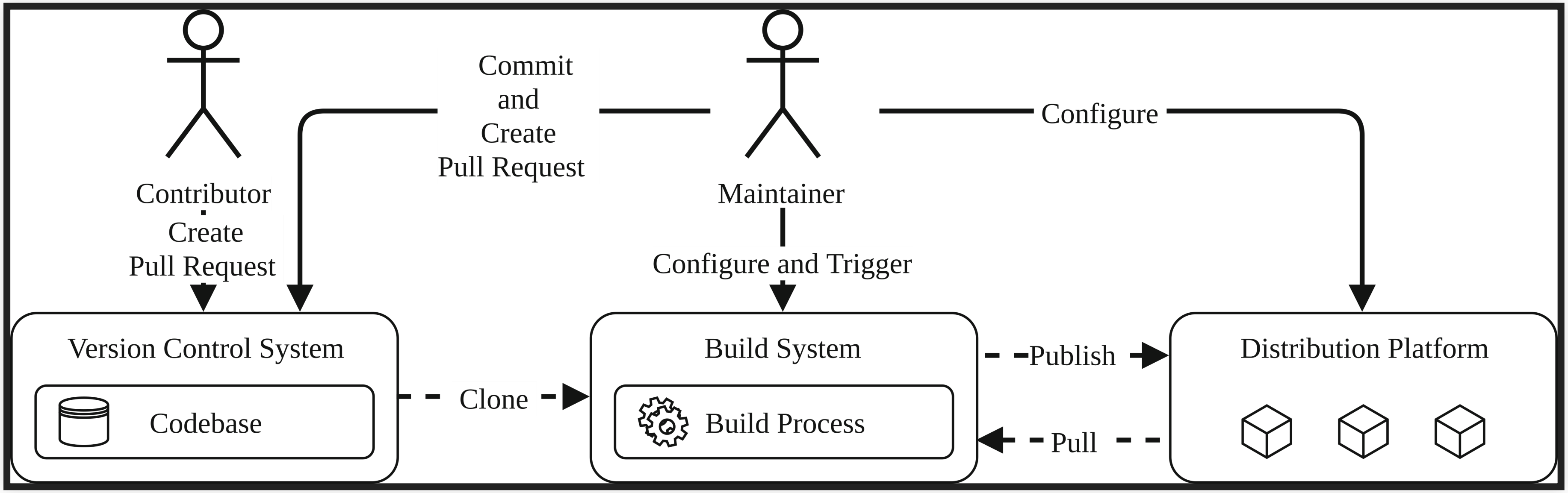
```
function leftpad (str, len, ch) {
    str = String(str);
    var i = -1;
    if (!ch && ch !== 0) ch = ' ';
    len = len - str.length;
    while (++i < len) { str = ch + str; }
    return str;
}
```

- Maintainer: *“I think I have the right of deleting all my stuff”*. “Unpublish” package.
- Impact: “many thousands of projects”—including major ones like babel and atom—no longer installable.
- NPM operators forcibly “un-unpublish” package.

Open source supply chain attacks (cont.)

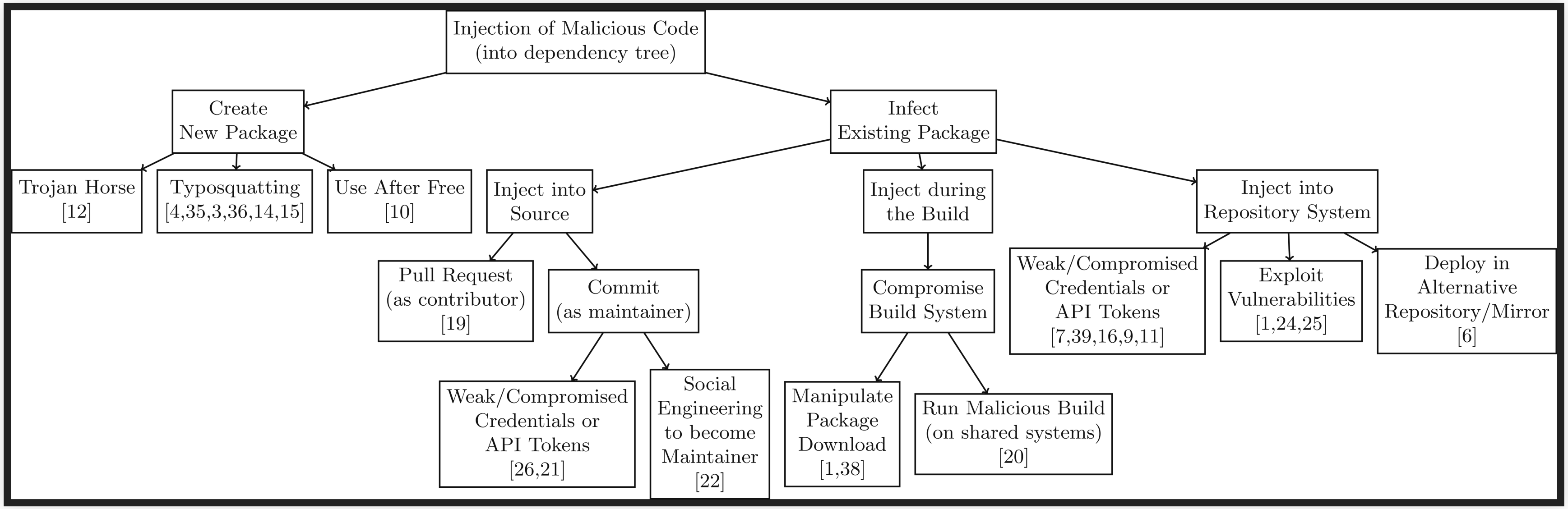
- For an attacker, code injection into (transitively) popular leaf packages has a **low opportunity cost**.
- Also, entirely open FOSS package ecosystems (≠ Linux distros) can be **easy to infiltrate**.

(An) open source development workflow



(image from [Ohm20])

Attack tree — Injection



(image from [Ohm20])

Attacker's goal: package P containing malicious code is available from download from a distribution platform *and* P is a reverse transitive dependency of a legitimate package.

Attack vector — Typosquatting

Injection → Create New Package → Typosquatting

1. Create a **new package** with a **name similar** (e.g., Levenshtein distance <= 2) to an existing popular package, including malicious code. Examples:
    - squat on PyPI the Debian package name (“python-sqlite” v. “sqlite”)
    - English variants (“color” v. “colour”)
    - Unicode tricks
2. Upload it to a distribution platform (e.g., PyPI)
3. Wait for users to mistype (e.g., `pip install python-sqlite`)

Related attack vector: **Use After Free**

Attack vector — Become maintainer

Injection → Infect Existing Package → Inject into Source → Commit (as maintainer) → Social Engineering to become Maintainer

1. Package maintainer: “I no longer have time for this project, who wants to take over its maintenance?”
2. Attacker: raises hand
3. Attacker: releases new version including malicious code

Might require early investment by the attacker to accrue enough “street credibility” to win over maintenance at the right moment. For popular packages with low bus factor it could be worth it.

Attack vector — Compromise build system

Injection of Malicious Code → Infect Existing Package → Inject during the Build → Compromise Build System

- Often, code run by users is *written but not built* by maintainers
- Rather, it is built by **3rd-party vendors**
    - e.g., GNU/Linux distros, app store operators, arch “porters”
- It hence becomes attractive to **break into vendor build systems**, compromising binaries “downstream”, without anybody auditing source code noticing

Related attack vectors: **Inject into [Package] Repository System** (≠ VCS)

Reproducible Builds

https://reproducible-builds.org/

[Lamb22]: Chris Lamb, Stefano Zacchiroli. *Reproducible Builds: Increasing the Integrity of Software Supply Chains*. IEEE Softw. 39(2): 62-70 (2022).

On untrusted code

> *“You can’t trust code that you did not totally create yourself. […] No amount of source-level verification or scrutiny will protect you from using untrusted code.”*
> *— Ken Thompson, Reflections on Trusting Trust, Turing Lecture 1984*

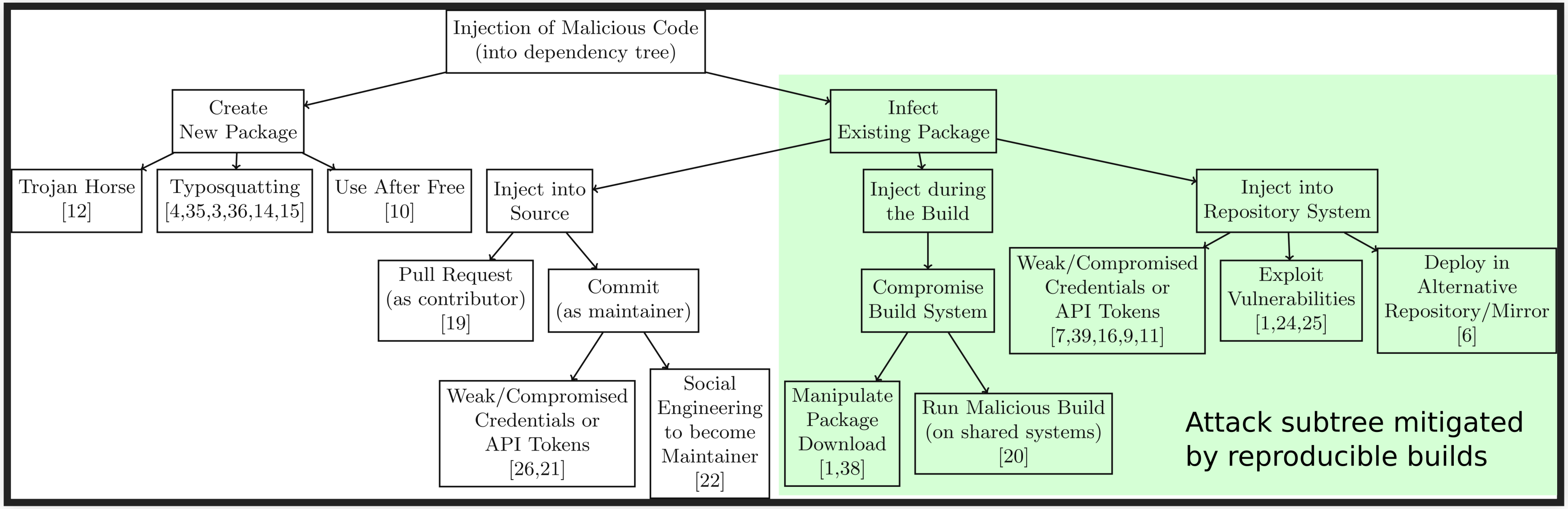
- 40 years later nobody “totally creates” code they run
- Reuse of open source software (FOSS) is everywhere in IT
    - “99% of audited code bases contain FOSS components” (Synopsis, 2020)
- Also, the FOSS *we run* is often not *built* by its developers

Problem statement

How can we increase users' trust when running (trusted) FOSS code built by (untrusted) 3rd-party vendors?

Problem statement

How can we increase users' trust when running (trusted) FOSS code built by (untrusted) 3rd-party vendors?



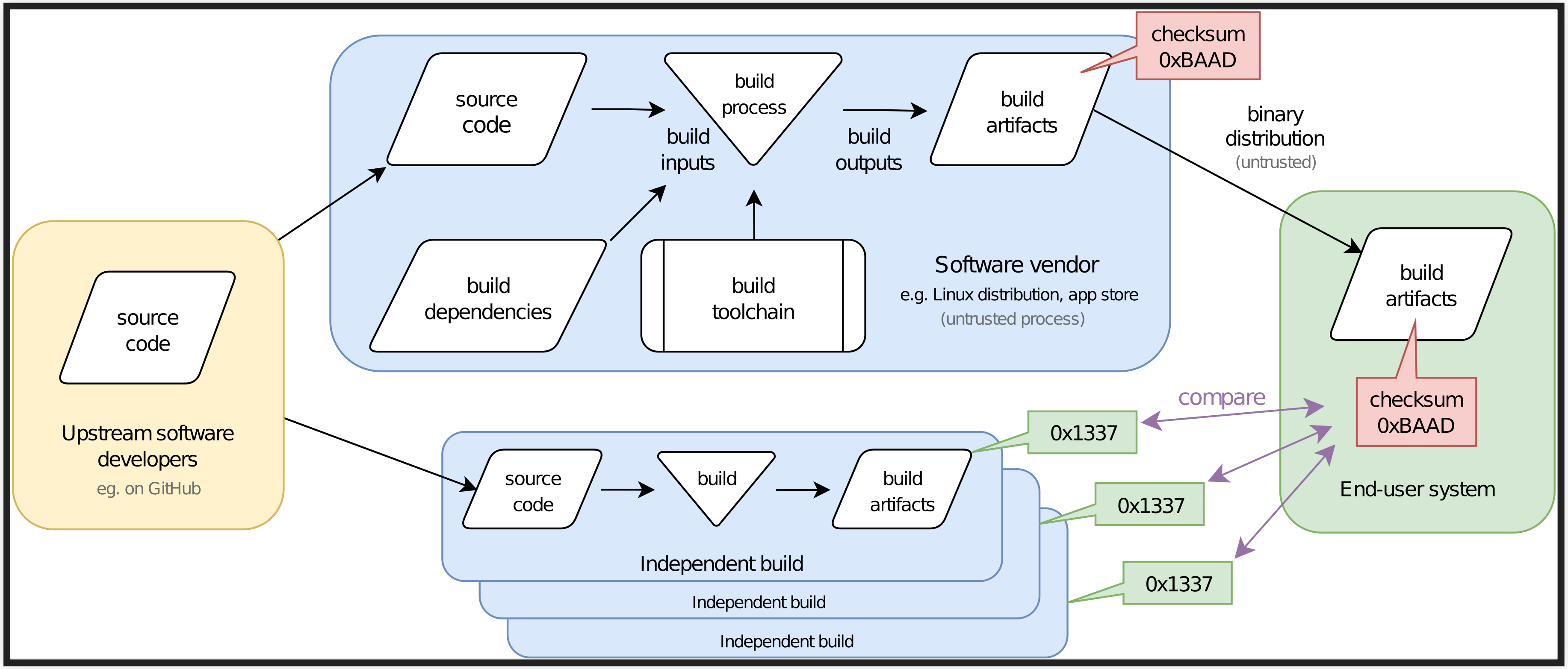
A reproducible build (r-b) process

Precondition/hypothesis: we can “reproducibly build” all relevant (FOSS) products, i.e.:

> *The **build process** of a software product is **reproducible** if, after designating a specific version of its source code and all of its build dependencies, every build produces **bit-for-bit identical artifacts**, no matter the environment in which the build is performed. — [Lamb22]*

(we’ll verify later how realistic this is)

R-B approach


checksum
0xBAAD
source
code
build
process
build
inputs
build
outputs
build
artifacts
binary
distribution
(untrusted)
build
dependencies
build
toolchain
Software vendor
e.g. Linux distribution, app store
(untrusted process)
source
code
Upstream software
developers
eg. on GitHub
build
artifacts
checksum
0xBAAD
compare
0x1337
source
code
build
build
artifacts
Independent build
0x1337
Independent build
0x1337
Independent build
End-user system

Making Debian reproducible

- Let's try a large-scale experiment: *making all Debian packages build reproducibly* from source
  - Debian: one of the largest and most popular GNU/Linux distro, esp. in the server/cloud market
  - 30'000+ (source) packages, 1+B lines of code
  - Initial goal of the reproducible-builds.org initiative, est. 2014
- Goals:
  1. Empirical experiment to identify common causes of non-reproducibility
  2. Real impact (if successful) due to Debian popularity in the market

Build reproducibility in the small

How hard could it be to ensure build reproducibility?

Build reproducibility in the small

How hard could it be to ensure build reproducibility?

After **controlling for source code, build deps., and toolchain**, two main classes of issues arise in practice:

1. **Uncontrolled build inputs:** when toolchains allow the build process to be affected by the surrounding environment.
    - Intuition: this is the build engineering equivalent of *breaking encapsulation* in programming
2. **Build non-determinism** that gets encoded in final built artifacts.

Build reproducibility in the small

How hard could it be to ensure build reproducibility?

After **controlling for source code, build deps., and toolchain**, two main classes of issues arise in practice:

1. **Uncontrolled build inputs:** when toolchains allow the build process to be affected by the surrounding environment.
   - Intuition: this is the build engineering equivalent of *breaking encapsulation* in programming
2. **Build non-determinism** that gets encoded in final built artifacts.

Let's see a bestiary of real-world examples…

Build timestamps

```c
void usage() {
    fprintf(stderr,
            "foo-utils version 3.141 (built %s)\n",
            __DATE__);
}
```

- The `__DATE__` C preprocessor macro “expands to a string constant that describes the date on which the preprocessor is being run.”
- Fix: `SOURCE_DATE_EPOCH` environment variable (standardized by r-b) to enable controlling for this

Build paths

```
fprintf (stderr,
    "DEBUG: boop (%s:%s\n",
    __FILE__, __LINE__);
```

- The `__FILE__` C preprocessor macro “expands to the name of the current input file”. This results in non reproducibility when the program is built from different directories, e.g., `/home/lamby/tmp` vs. `/home/zack/tmp`.
- Fix: introducted gcc `-ffile-prefix-map` option (and related `-fdebug-prefix-map`) to support embedding relative (rather than absolute) paths

Filesystem ordering

```
NAME
    readdir - read a directory

SYNOPSIS
    #include <dirent.h>
    struct dirent *readdir(DIR *dirp);

[…] The order in which filenames are read by successive calls to
readdir() depends on the filesystem implementation; it is unlikely
that the names will be sorted in any fashion. […]
```

- Fix: impose a deterministic order in build systems/recipes, e.g., via an explicit `sort()`

Archive metadata

- Archive formats like `.zip` and `.tar` embed various kinds of metadata by default
    - User/group ownership (e.g., `zack` v. `lamby`)
    - File modes (umask)
    - Timestamps
- Fix: control for this, e.g.:
    - `tar --owner=0 --clamp-mtime=T`
    - `touch --date=$SOURCE_DATE_EPOCH`

Randomness

Even when the entire environment inputs are controlled for, many builds remain *non-deterministic*. For instance due to **randomness in unexpected places**.

```
my %h = ( a => 1, b => 2, c => 3);
foreach my $k (keys %h) {
    print "$k\n";
}
```

- Perl’s hash type does not specify an ordering for key traversal, so a call to `sort` should be inserted before `keys %h` to make it deterministic.

Uninitialized memory

- Many data structures contain **undefined areas** that do not affect their operation, but could end up being **serialized into build artifacts**.
- *Padding for natural memory alignment* can also be filled with arbitrary content.
- Fix: explicitly zero-out memory.

```
--- a/direntry.c
+++ b/direntry.c
@@ -24,6 +24,7 @@

  void initializeDirentry(
      direntry_t *entry, Stream_t *Dir) {
+         memset(entry, 0, sizeof(direntry_t));
          entry->entry = -1;
          entry->Dir = Dir;
```

- A patch for GNU mtools to ensure a `direntry_t` struct does not contain uninitialized memory.

Build reproducibility in the large

- Let’s now assume we know how to fix all micro-issues that affact build reproducibility.
- How do we go about **making large FOSS software collections reproducible**?
    - Use case: Debian
- Approach: establish a corresponding **Quality Assurance process** and soft-enforce it using Continuous Integration (CI).

Adversarial rebuilding

How do you find build reproducibility issues, at scale?

- Mass-rebuild all packages…
- …building each of them twice…
- …in two build environments configured to **differ as much as possible**:
    - Clock set 18 months in the future in 2nd build
    - Changing: hostname, locales, kernel
    - Reverse filesystem ordering using disorderfs
    - 30+ variations in total

Recording build information

- According to our definition of a reproducible build, *legitimate build inputs* should be controlled for and replicated identical in the 2nd build:
    - Source version of product under build
    - Ditto for all transitive build dependencies
    - Toolchain version
- To that end, the `.buildinfo` file format has been designed to capture these information

`.buildinfo` — Example

```
Source: black
Version: 20.8b1-1
Checksums-Sha1:
  9915459ae7a1a5c3efb984d7e5472f7976e996b1 2584 black_20.8b1-1.dsc
  14bfd3011b795f85edbc8cc4dc034a91cfaa9bcd 111096 black_20.8b1-1_all.deb
  69c3d4ae7115c51e7b00befe8b4afd5963601d66 285684 python-black-doc_20.8b1-1_all.d
Checksums-Sha256: [...]
Build-Architecture: amd64
Installed-Build-Depends: autoconf (= 2.69-11.1), automake (= 1:1.16.2-4), […], gc
```

- An example `.buildinfo` file, recording both the environment and results of building Debian's `black` package. (See full version.)

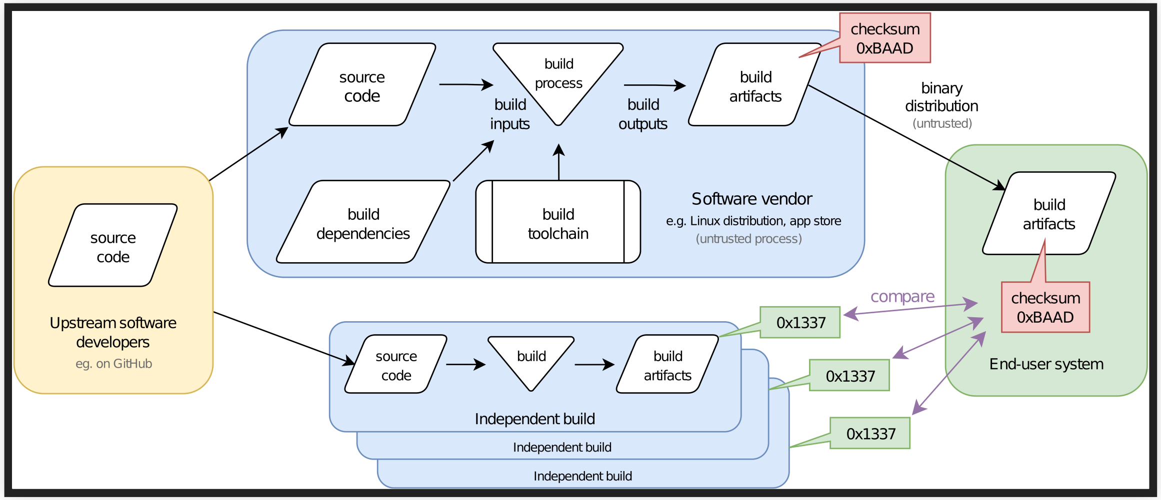
Build attestations

- `.buildinfo` files also contain the *cryptographic checksums of final build artifacts*, making them fill the role of **build attestations**:

> *I, Alice, given source X, build dependencies Y_1,...,Y_n and toolchain Z, have conducted a build run obtaining a set of artifacts with checksums K_1,...,K_m.*

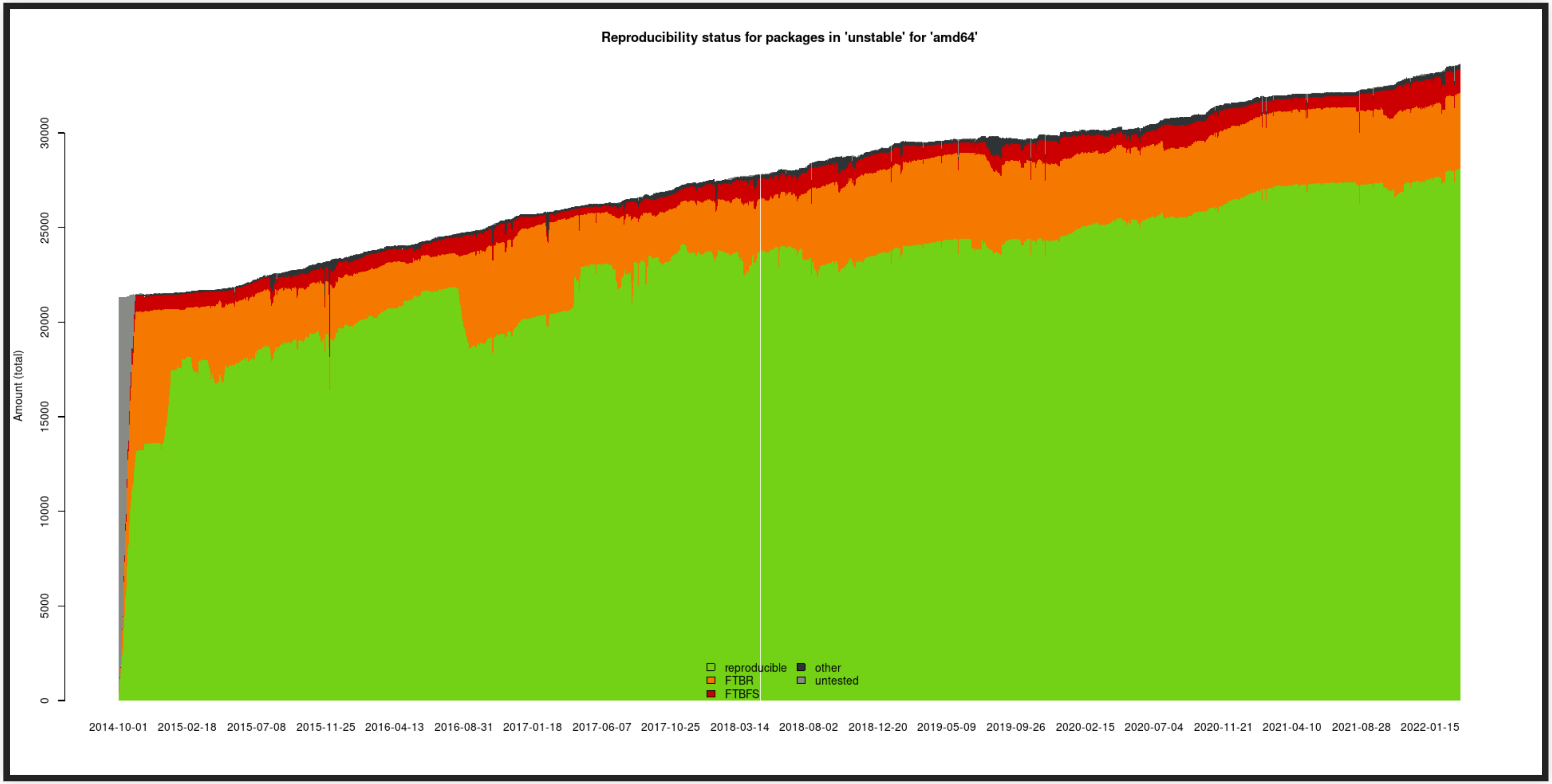
- Anyone (for QA or independent verification purposes) can rerun the build and publish their own build attestations

.buildinfo — Usage



- Before installation, users verify package checksums against published build attestations
- Published by either vendors they trust; or relying on some consensus within a network of independent rebuilders
- Debian publishes 27+M build attestations at https://buildinfo.debian.net

Reproducible Debian — Evolution over time



Quality assurance synergies

- Systematic R-B testing ⇒ systematic **build testing**, catching any FTBFS (Fail To Build From Source) bug
- Some software will only FTBFS in the extreme R-B build environment; fixing it will make the **software more robust** in general
    - E.g., expired SSL certificates at +18 months, or unusual timezone offsets
- R-B testing can detect user-level breakages by serendipity
    - E.g., HTML documentation pointing to `/tmp/build/foo/usage.html` instead of `/usr/share/doc/foo/usage.html`

Quality assurance synergies — security

- Security issues can also be spotted during R-B testing by serendipity

```
{
  'cgibin' => '/usr/lib/cgi-bin/gbrowse',
  'conf' => '/etc/gbrowse',
  'databases' => '/var/lib/gbrowse/databases',
  'htdocs' => '/usr/share/gbrowse/htdocs',
  'OpenIDConsumerSecret' => '639098210478536',
  'tmp' => '/var/cache/gbrowse'
},
```

- An example `ConfigData.pm`. As it was created at build time, all users shared the same `OpenIDConsumerSecret`. (See: Debian bug #833885.)

The Reproducible Builds ecosystem

https://reproducible-builds.org/

- 2014: project started by Debian developers for ~~Debian needs~~ fun
- Joined since: Arch Linux, coreboot, F-Droid, Fedora, FreeBSD, Guix, NixOS, openSUSE, Qubes, Tails, …
- 2017 milestone: Tails (live distro used by Snowden to exfiltrate NSA documents) publishes a fully reproducible ISO to improve end-user verifiability
- R-B is an independent project hosted by Software Freedom Conservancy and supported by 3rd-party sponsors (e.g., Google, The Linux Foundation, Ford Foundation, Siemens)

Challenges

- Debian reached 95% reproducible packages, can we go all the way?
    - Yes, it’s just busy/constant maintenance work.
    - Working with upstream and spreading r-b culture helps a lot.
- How to make *signed buld artifacts* reproducible (without distributing signing keys)?
    - Detached signatures. (Painful for distribution channels.)
- How do end-user verify build artifacts before installation?
    - Particularly challenging on locked-down mobile environments/stores.
- How little trusted code is acceptable?
    - Bootstrappable Builds managed to bootstrap from a 6 KiB trusted ELF binary to GCC via TCC.

Takeaways

- **Open source software supply chain attacks** are both a big issue and a hot topic in cybersecurity right now.
- **Reproducible Builds** help countering *build/distribution injection* attacks, by enabling (distributed) anti-tampering detection at the executable/package level.

Learn more

- Chris Lamb, Stefano Zacchiroli. *Reproducible Builds: Increasing the Integrity of Software Supply Chains*. IEEE Softw. 39(2): 62-70 (2022).

Advertisement

Télécom Paris is **hiring a tenured associate professor on this topic**! (and *software engineering for cybersecurity* more generally). Check my home page (https://upsilon.cc) or ask me for details.

Appendix

Root cause analysis — Diffoscope

dolfinx-doc_2019.2.0~git20200128.797071f-3_all.deb 1.16 KB

file list 367 B

Offset 1, 3 lines modified		Offset 1, 3 lines modified	
1	-rw-r--r-- 0 0 0 4 2020-02-03 15:41:41.000000 debian-binary	1	-rw-r--r-- 0 0 0 4 2020-02-03 15:41:41.000000 debian-binary
2	-rw-r--r-- 0 0 0 1664 2020-02-03 15:41:41.000000 control.tar.xz	2	-rw-r--r-- 0 0 0 1664 2020-02-03 15:41:41.000000 control.tar.xz
3	-rw-r--r-- 0 0 0 190104 2020-02-03 15:41:41.000000 data.tar.xz	3	-rw-r--r-- 0 0 0 190100 2020-02-03 15:41:41.000000 data.tar.xz

data.tar.xz 625 B

data.tar 603 B

./usr/share/dolfinx/demo/hyperelasticity/CMakeLists.txt 587 B

Offset 10, 15 lines modified		Offset 10, 15 lines modified	
10	# Get DOLFINX configuration data (DOLFINXConfig.cmake must be in	10	# Get DOLFINX configuration data (DOLFINXConfig.cmake must be in
11	# DOLFINX_CMAKE_CONFIG_PATH)	11	# DOLFINX_CMAKE_CONFIG_PATH)
12	if (NOT TARGET dolfinx)	12	if (NOT TARGET dolfinx)
13	find_package(DOLFINX REQUIRED)	13	find_package(DOLFINX REQUIRED)
14	endif()	14	endif()
15	# Executable	15	# Executable
16	add_executable(${PROJECT_NAME} hyperelasticity.c main.cpp)	16	add_executable(${PROJECT_NAME} main.cpp hyperelasticity.c)
17	# Set C++17 standard	17	# Set C++17 standard
18	target_compile_features(${PROJECT_NAME} PRIVATE cxx_std_17)	18	target_compile_features(${PROJECT_NAME} PRIVATE cxx_std_17)
19	# Target libraries	19	# Target libraries
20	target_link_libraries(${PROJECT_NAME} dolfinx)	20	target_link_libraries(${PROJECT_NAME} dolfinx)