Improving Build Time SBoMs

Joshua Watt

Linux Plumbers

December 11, 2025

yocto PROJECT

About Me

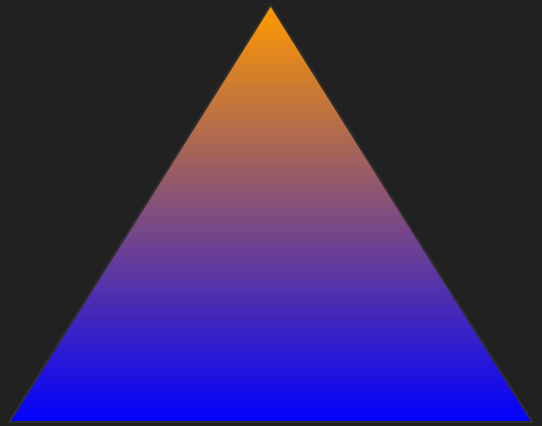
- Worked at Garmin since 2009
- Using OpenEmbedded & Yocto Project since 2016
- Member of the OpenEmbedded Technical Steering Committee (TSC)
- Member of Yocto TSC
- Member of the SPDX Technical Team
- Joshua.Watt@garmin.com
- JPEWhacker@gmail.com
- IRC (OFTC or libera): JPEW
- X/Twitter: @JPEW_dev
- LinkedIn: joshua-watt-dev

What is a build time SBoM

Build Time SBoM

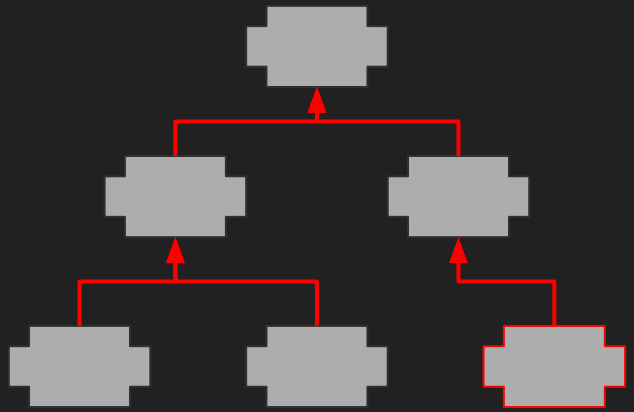
- Documents a build *process* that occurred at a given point in time
- Inputs to that process
- Outputs from that process
- Attestable

Sliding Scale, but also Composable



High level user commands

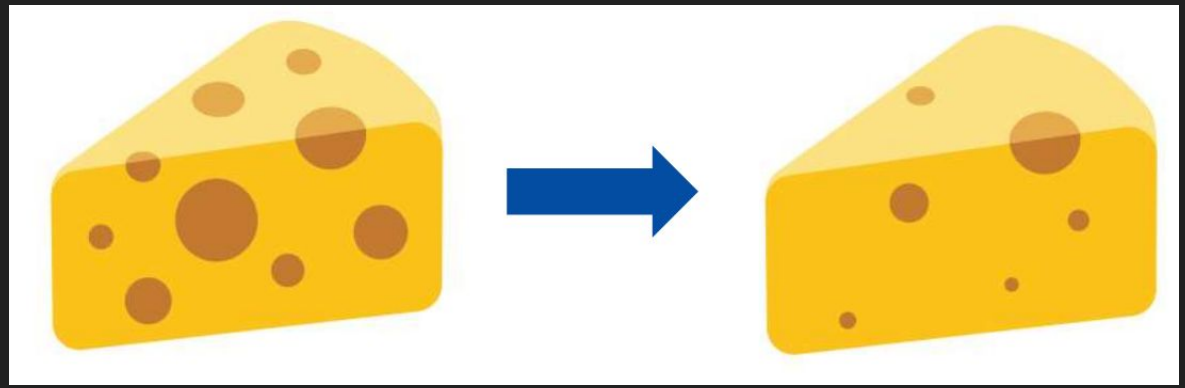
Individual compiler invocations



Why Build Time SBoMs?

Regulatory Compliance

- Most SBoM requirements are focused on the "runtime" deliverables, not necessarily transiting into "built time" dependencies
  - E.g. What code is running, not necessarily how that code came to be
  - The most comprehensive requirements that talk about built time are probably BSI TR-03183-2
- This seems likely to change in the near future
- Unknown what the CRA will require



https://www.cisa.gov/sites/default/files/2023-09/EU%20Commission%20SBOM%20Work_508c.pdf

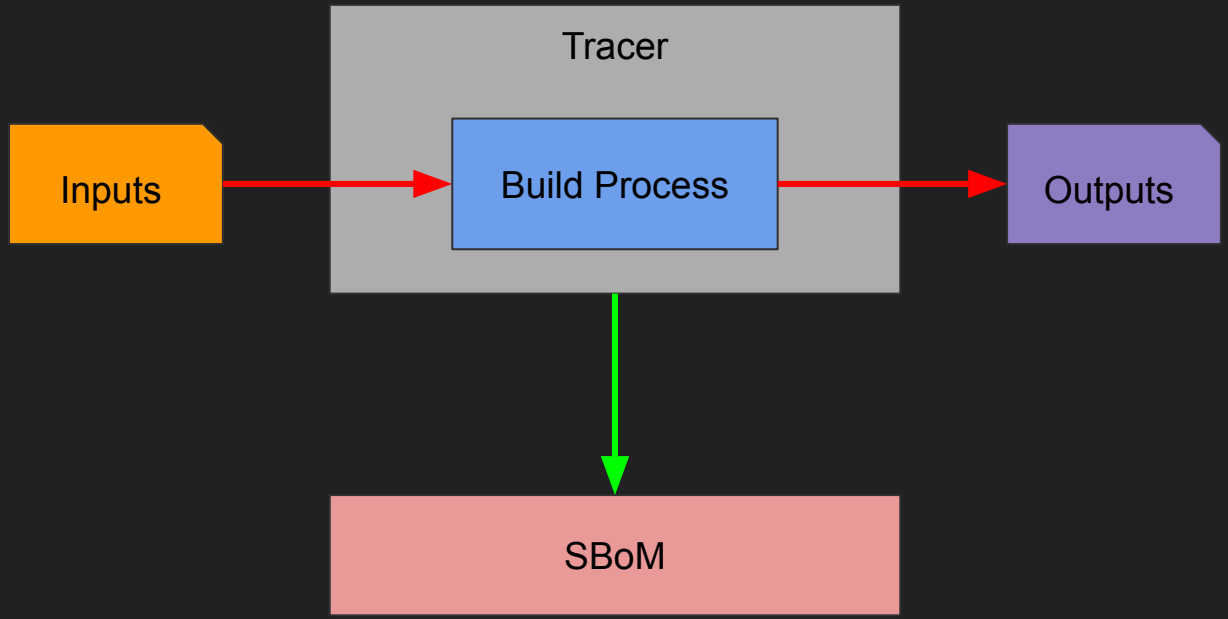
The build time software supply chain is important!

- High profile software supply chain compromises (Solar Winds, xz-backdoor, etc.)
- Building your own software from source means you need to track your software supply chain
  - Binary hashes likely won't match
- Even if you consume pre-built binaries, *someone* built that binary; how can we trace its lineage?
- The desire to provide attestation for every build step is on the rise (e.g. SLSA)

Prior Art

Tracing

- Trace build processes (e.g. using strace or similar) and produce SBoM from the trace



Tracing

Pros:

- Complete
- Toolchain Agnostic
- More flexible to run on unknown build processes

Cons:

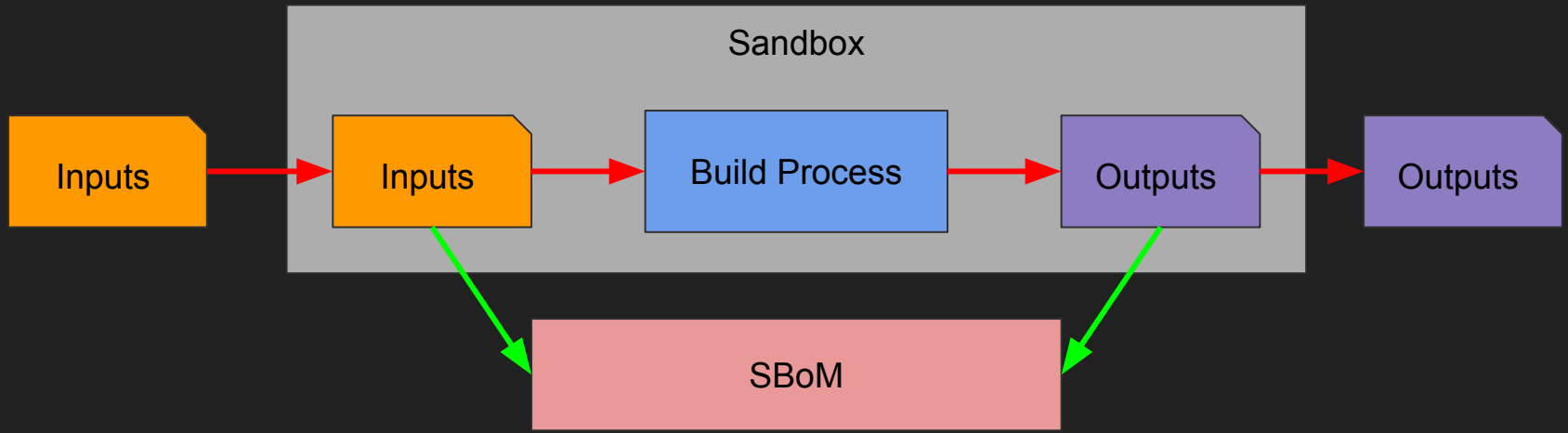
- High Overhead
- May need elevated privileges
- Lack context (not all files are *necessarily* build inputs)

Examples:

- SBoMit

Sandbox Approach

- Track build inputs (dependencies, tools, etc.) and outputs
- Build in a sandbox environment to ensure that all inputs and outputs are accounted for



Sandbox Approach

Pros:

- Medium Overhead (Efficient copy is key)
- Generally desirable in many contexts anyway (e.g. cross-compiling)
- Toolchain Agnostic

Cons:

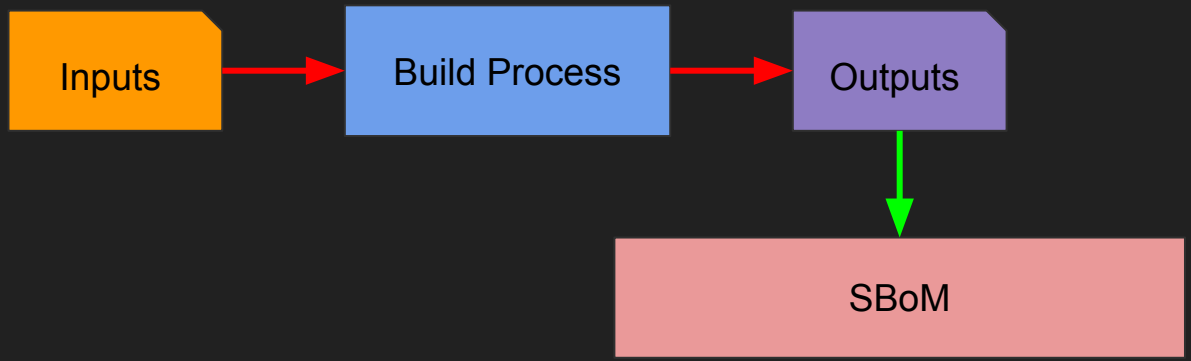
- Over-estimates inputs (unused inputs are included)
- Complex to setup sandbox environment

Examples:

- Yocto Project

Post-Build Scan

- Scan (or execute) outputs after they are built



Post-Build Scan

Pros

- Efficient
- Flexible (e.g. able to be integrated into many pipelines)
- Simple to integrate

Cons

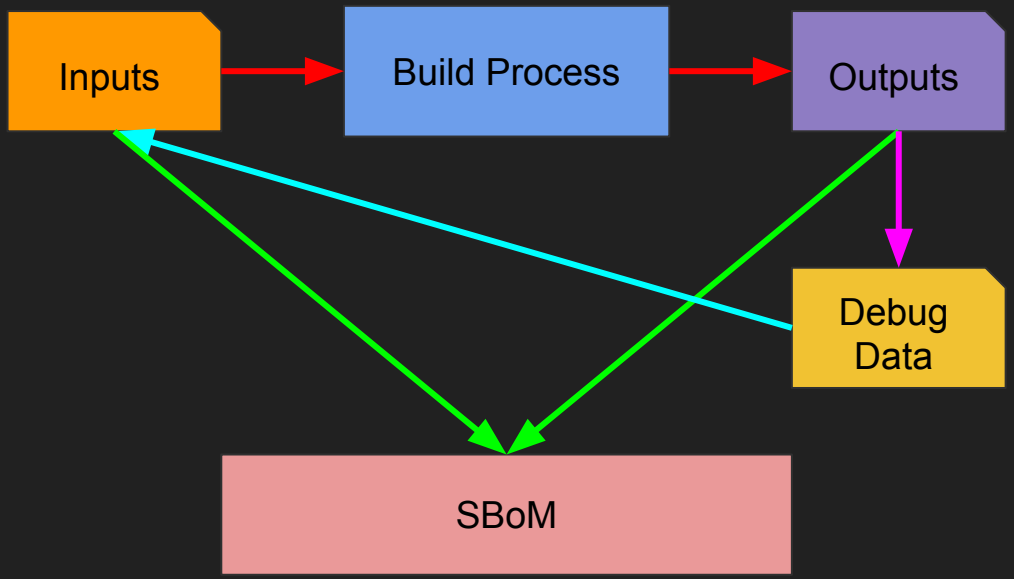
- Execution may not be desirable or possible (e.g. cross-compiling)
- Misses static dependencies
- Language specific?

Examples

- Heimdall

Debug Info

- Extract input files from debug data in binaries (e.g. DWARF debug data)



Debug Info

Pros:

- Extracted from data already produced during build
- Quick to extract
- Accurate (e.g. can even detect static library code)

Cons:

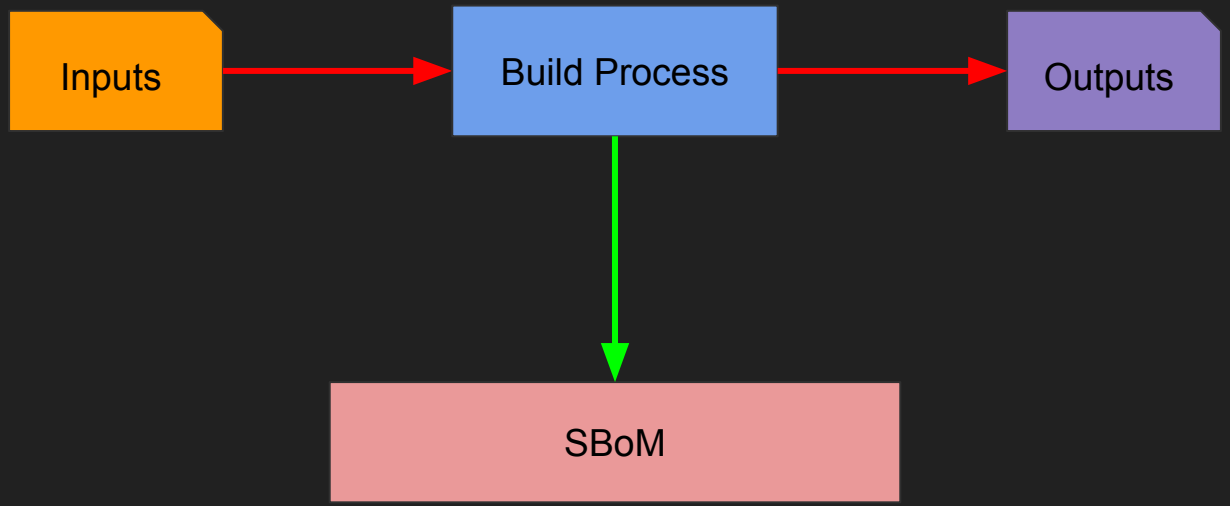
- Unintended use of debug data?
- Not useful for files without debug data
- Dependent on compile time options (no debug data, no SBoM)

Examples:

- Yocto Project
- Heimdall

Toolchain Feature

- Modify toolchains to produce supply chain information



Toolchain Feature

Pros:

- Accurate and Complete information
- Efficient

Cons:

- Requires each toolchain to implement changes to produce information

Examples:

- Esstra
- Heimdall

Discussion