Filecoin - warpfork Q&A

Hi! I see some enthusiasm here that I love

Let's say I'm someone who's already 10,000% sold on the value of reproducible builds and reproducible interactions with package management systems (I am!) and also already 100,000% sold on the value of doing it with content addressed data and data-structures (I am!). But also a tad unfamiliar with python packaging ecosystems :) and so could use some more background and connective information about how this project hopes to impact that space specifically.

- *Can you give me a little more run-down and comparison to alternative systems?*
  - *I know there's a lot of package systems in python, so a quick table or even just bulleted list of quickcompares would be immensely useful for helping readers understand why a new project is needed!*
  - *This project description jumps pretty directly to describing itself in terms of Conda. Why Conda? Why about Conda channels makes them recurring concept in the description?*
    - *Assume I don't know python packaging ecosystems -- how do I explain this choice to another person who also isn't familiar with python packaging ecosystems? (Even assuming we both care about the domain and outcomes!)*
    - *Will this project depend on the continued existence and community of Conda to continue containing up-to-date information itself?*
    - *Will this project "export" concepts from Conda to its downstream consumers? Will I need to read Conda docs to understand it, or use Conda tools to consume it?*
  - *I see pip and pypi referred to also, but again, if the audience isn't an every-day-is-python developer, the map here is hazy and would benefit from more explanation :)*

I took some time recently to experiment with poetry, hatch and packaging in response to this inquiry as I have been a user mainly of conda and pip (and some of the environment management tools downstream of pip) and of course docker as a kind of "everything" container, including its use in the python ecosystem as a way of providing blanket isolation for dependency systems, so I can briefly describe what I believe are the most popular systems for managing python app dependencies and dealing with python apps written in the most common style.

The background, the how and why of python, setuptools, pip and pypi

First, in the end, python is just an interpreter for a programming language, and as such it's written with a fairly conservative set of capabilities for importing and using code in named 'modules' and hierarchies of modules. From the earliest versions I'm aware of, python modules came in two flavors; either they were just python files with names that could be located through a path search, looking in the PYTHONPATH environment variable, directories containing python files and an__init__.py, or shared libraries (dll, so, dylib) on the platform python is running on that export some basic symbols allowing python to create a PyModuleDef struct for its native code.

At this time, setuptools and a standard package set for the python ecosystem didn't exist; egg and wheel, file formats used for python software distribution didn't exist and when you wanted to install a python app, you'd either get its dependencies from the operating system's package manager in case you were a wierdo running linux, (more on this later) or more commonly download and individually install each of the packages using its own installer or by having a script that created a large PYTHONPATH with all the installed packages in it before running whatever thing you were interested in. People made some packagers in the early days that amounted to zipping up a bunch of things that could be found in the PYTHONPATH and making either a standalone exe that links libpython or a shell script that sets the path and invokes python on the right source file.

In those days people were used to heavyweight software installers that copied a lot of data, weren't used to having a lot of OS level dependencies other than shared libraries with C ABI functions exported and the licensing model of open source vs closed source was treated in many ways as fluidly as it is today, unlike many of the years in between; with electron and npm dependencies, closed source apps can exist in a middle state that depends on a lot of open source software but is treated as partly closed source, and a good many apps are distributed this way nowadays; so too in the 90s when python got its start, app distribution was often done this way. Python programs often existed in a situation where they depended on specific versions of the ecosystem or specific exports from programs that they were intended to run in. The "linking exception" from the LGPL

also factored in; closed source programs that were driven by TCL, perl and python often shared space with properly open source libraries and open source python in this way.

For the computer environments of the era, this wasn't the worst way of doing things. Languages and modules evolved slowly and open source software was more of an exception than the rule it is today. setuptools, the python library used to resolve dependencies, install python modules from pypi and similar repositories and the repositories themselves, along with ways of advertising these libraries as separately installable; entities of their own didn't properly exist and few were using them that way, often picking them apart or using the day's packagers to agglomerate them into full apps. That system though, as use of python grew, python programmers came to expect the environments they'd been using and the reach of python code itself grew, the fracturing of the python ecosystem began to become a problem for python programmers.

OS distributions at first packaged a single version of python and upgraded it periodically. In the early days, interfaces remained consistent and feature testing in python code (similar to conventional wisdom methods of writing robust javascript code that would grow up about a decade later), were enough for python apps in the nascent ecosystem to function across those upgrades. Libraries were a bit tougher, but OS distributions often patched in a full set of libraries for the ecosystem in their tools (rpm, deb, ebuild etc). This worked ok for a while, but it was still very common in those days for a random python app one wanted to run to have peculiar interactions with the base system and need special setup.

OS distributions went through a phase of first giving choices, then slotting packages of all kinds (leading to the need for slotting to be woven into things that used packages downstream, something that really isn't the concern of most of those things). Slotting is when an OS distribution provides multiple variants of a package by different names, such as allowing python 2.7 and 3.9 via the names 'python2' and 'python3'. Since software build systems are automatic, various tricks were used to try to make slotted packages work well in an ecosystem that didn't support it, but ultimately it was a failed strategy to make library code slotted when languages and the packages themselves were opinionated about which version was the one that would be named without a version number.

It is in this era when setuptools grew up in python, having seen the success of perl's CPAN and its legacy in allowing the interchange of higher level library code among perl programmers, python gained an "indexing service" and a metadata format for package distribution that python's tools could use to download packages. The web service does a good job at what it was designed to do; make python packages easy to distribute even when one didn't have a sophisticated web service; it's based on the directory service formats of the day and normal html hyperlinking, in some ways echoing modern REST style hypermedia, but this was before Douglas Crockford formalized JSON, in an era where our expectation was more of XML and the bespoke family of angle braces schemas as a lingua franca interchange format.

Some things that are expected today, such as idempotence of uploads and hashes as part of verifying downloads were not included. End users generally weren't using pypi by itself but it was expected to be developer experience like npm.

Attacks on the open source ecosystem, big businesses relying on frequently updated libraries and the like were not common in this era so few were considering how that kind of infrastructure could be subject to attack. Most were very idealistic in this era about the openness of publishing packages, and it felt a bit subversive.

Many in the mainstream had a negative view of open source software in general, as evidenced by Microsoft's contemporary "get the facts" campaign.

So it's into this environment that python's pypi launched, and it works well for what it is. A well specified dependencies entry in setuptools can allow a python program to set itself up quickly; individual user directories were added so that pip users wouldn't have to disturb the OS level package install (probably lagging considerably behind on behalf of its own flagship python apps, such as Redhat's anaconda).

Cracks form as the weight and breadth of the python ecosystem take their toll.

Having a single set of installed packages per user is enough for someone whose deploy environments look like their laptops; (famously, the use case for freebsd in yahoo). When one works on a single project, has tools for that project installed in the path, and the same dependencies serve the app to be worked on, it can be a very cozy experience, but that isn't today's reality.

We often work in branches, with multiple versions of the same app and incompatible tools. We do code reviews and tests for others working ahead or behind or on versions otherwise incompatible and we need our own environments to stay as they are and not be mutated. We run ```pip install``` frequently in docker images and cloud setups and expect them to work the same, with the same inputs, time and time again as we distribute as small as possible source sets to our cloud instances.

Python in the mean time gained a sort of disconnected support for this environment sharding, the virtual env, which one can set up for pip and pypi use in various ways (there are a lot of virtualenv management tools out there). All of them cooperate with python in the same way, by carefully crafting shell scripts and environment settings that interact with python's module reader to read the code they've installed in a separate space serving only one or one set of uses. Poetry and conda also use this mechanism, although using their own machinery and own spin on what code these should resolve to. This is where we stand today with flask apps and services being deployed and being very connected to the code they depend on, and handling important commerce and safety information. Hashes have been added of the packaged files themselves, along with precompiled binaries, but the metadata format, being still the old hyperlinked format, doesn't allow one to validate when or how things changed, or even know if a specific version of a package is the same one one downloaded yesterday. You can keep that info yourself and use your own tooling to work with it, but in speaking with a good many devops people, most aren't going to such lengths and writing tools for this.

How package systems are used and how they compare

- The classic pip uses the default ```pypi``` ecosystem to draw from. It has basic version range constraint solving, but doesn't automatically remember its precise inputs and isn't opinionated about where packages go (sharded or into the user's global package set). Used in the normal way, it's eminently possible to receive a package that's been updated in place in the index without knowing it.
- Poetry, coming later, is built to mimic the more popular npm style also mimicked by cargo and PHP's composer. It still uses the pypi ecosystem but in the style of npm and others, drops a poetry.lock file that contains an exact description of the imported source. It by default installs packages in a directory-local shard below its pyproject.toml file like NPM does and like npm, provides a 'run' verb that runs one's code within poetry's notion of what it's runtime environment should be.
- Hatch takes inspiration from conda and npm, in that it operates in a directory local way like poetry, but provides an environment management interface that can leave the shell in a environment hydrated command line like conda, and these are primary use cases. Hatch's most important feature is its nix-like building of packages in isolated clean environments, giving it the ability to produce reproducible artifacts when the packages themselves allow that. It is an in-between in a lot of ways. Unlike pip it has comprehensive build control, unlike poetry it's more user experience focused and unlike conda it does local shard management.
- Conda is the gorilla of python package systems at the moment. It has a deep dependency and build manager supporting multiple ways of doing package resolution and building. Having grown up later, it supports comprehensive metadata that allows the user to record and inspect the entire ecosystem offered by set of channels and verify its outputs. It has environment management, but the environments by default are centralized although you can override it.

This proposal.

Since our proposal focused on backend metadata and verifiability, the conda formats are a better choice, already support a large base of users. By also producing an index that's compatible with the pypi ecosystem, every other way of satisfying python dependencies would be supported as well. Conda is a package index, a host for a package ecosystem build and its builder and the software needed to use these. The software part never need die as long as someone wants it and with it the formats it uses. Since most use the pypi package index (even conda users, using pip to install packages from the standard ecosystem), producing it is a good choice, and even if tomorrow everyone decides conda was a bad idea, the extra information produced and the tooling to produce a pypi compatible service capable of being used for package installs will endure as long as people want.

- ***I think it would be especially great to talk about what kind of data structures will be produced... And ideally, in terms of what new tools could do in navigating them.***
  - ***For example, I was recently at the Reproducible Builds Summit, and the community there talked at length about building "binary transparency logs" and "package release transparency logs", i.e. analogous to the transparency logs we nowadays have for TLS certs to make sure there's no misbehaviors or surprise-updates in the wild that would defy expectations.***
    - ***Will this project export data structures that might be desirable and easy to use to build such a (general, non-python-specific) PRTL service in the future?***

We didn't consider doing much but producing an rss-like listing of releases that could be compared. Conda's format has hashes for packages being released downstream and the upstreams would be copied into identifiable buckets, but it's not a bad idea to produce more.

What's here is already ambitious, though and this type of governance makes sense when there's more widespread use. Having persistent channel definitions in immutable storage is already a great improvement.

Reproducibility is an adjacent thing to what we're thinking of. It should be the case that we work toward a reproducible ecosystem build for each channel increment but more important is ensuring that the artifacts left behind by python's setuptools are reproducible for a given dependency set to be resolved. It's intended that that will be primarily achieved via a controlled package index based on specific snapshots of the ecosystem and tooling that enforces to the degree possible use of that index and resistance to circumventing setuptools' version resolver. it's intended that files within reach of the PYTHONPATH will be required to match those sourced from the selected ecosystem snapshot otherwise, the process will fail. it is intended that users will have the ability to determine specific path exceptions when undergoing this process so that unusual installs can still succeed with acknowledgement,

---

- ***Which of these goals are key, and which are stretch?***
  - ***Some examples of things I'm a little surprised by because they seem like large scope or quite tangental (unless there's a connection I've not intuited -- but then others could probably use the explanation too)...***

- ***"sandbox runner based on libseccomp" -> as someone who's worked in this area before, this sounds like a very big subtask :) How critical is this? Is there something off-the-shelf you will use? What happens if you don't do this, or it turns out to be a much bigger task than you expected, or turns out to be fragile even when completed, etc?***
- ***What's the relevance of BitTorrent BEP-30?***

I've cut it down to essential goals for the most part and sorted in order of need, so everything depends on stuff that's prior in this list. A few things can be optional. We don't strictly need a sandbox to just provide ecosystem snapshots people can opt into. I've used libsandbox (as in gentoo) before and made some of this in a different form in there, which could be an adequate fallback, or the project could do fully without. I think it's important in that, as I mentioned, many python packages run live code and try to be smart about how they're installed (some even going so far as to use pip to uninstall or upgrade things they find problematic). Using a sandbox to ensure that these efforts are flagged and dealt with in some way is a good step toward knowing what's installed and ensuring that it gets reproduced correctly every time. Muinin's other code, not open source as of now would be available to the person muinin uses for this along with access to my knowledge of the subject of sandboxed builds. While I'm not an expert in seccomp itself, I have fair expertise with infiltrating build processes to lock down and supervise the process itself, and the knowledge i have, as well as what muinin crystallized while investigating a reproducible build enforcer as a product can be passed on.

I referenced BEP-30 because it gives a nice provable format for a container of files and directories that have proven content (and further, the BEP-30 format is supported by bittorrent clients, allowing seamless mirroring outside the IPFS ecosystem. What would be conferred by creating an archive of the full ecosystem built into the channel in that would be the ability to systematically check that downloads were provided correctly without having to write any external or bespoke tools, and the artifact of the torrent describing the full archive would be useful in itself. Other formats exist for describing built code for repeatable builds, but it's often surprising what they lack; either a single unified hash format for the inputs or products; often not all inputs are described, such as patches that are applied during the build process, sometimes either the inputs or outputs aren't fully described and usually there's no tool or only somewhat dodgy shell scripts for fully checking the contents (and almost always targeted at the project's own CI, not all at once for end user consumption).

In short, it's a well considered, durable and checkable archive header that one can think of as being useful both as a description of a set of data and a way of mirroring that data, so it fulfills a good many purposes in such a system even if I haven't thought of everything it might be useful for.

- ***Generally I think it would be nice to hear quite a bit more about potential risks and how to manage them, beyond "people don't use it" :) Why would that happen? Can we think of some reasons, and try to address them in advance (or even dismiss them, but knowingly)?***

It's unlikely this project will have the bandwidth to do a lot of custom work making the ecosystem do anything in particular, but if there's a real need for building in custom setuptools or distutils (for example), people would be needed to do that and there's a general risk that someone on the team will take a future world crisis as an opportunity to use the index and its packages as protestware.

It could be infiltrated by north korean state actors if people aren't careful about hiring I suppose or a rushed process under pressure could take low quality inputs or attacks (such as coin mining or credential stealing) code without knowing it. The attempt being made here is to ensure that the ecosystem provided in each slice is better than the original and that captures of the full slice allow packages in the index to work together but there will always be people using it in ways we can't predict and employees aren't always predictable either.

Since it's served from ipfs, some will be using it via ipfs gateways and those have a chance of rewriting history in ways that won't be observable if the attack is well enough coordinated. A user is better served running a local ipfs node but where they get that code can also be subject to attack. There are chicken and egg problems all the way down obviously, but those are already a problem. If someone can attack your ipfs node, they can easily silently enable insecure SSL in your http libraries as well.

It's possible somebody will hear "it's a safer alternative to pypi" not really knowing what's going on, set up their global proxy to point to a channel provided by this project and use pip from there on, thinking they're getting updates when those need to be opted into by selecting a later set. Someone could be very confused (and maybe upset) by manipulating their local python install from inconsistent indices... Maybe somebody painstakingly tricked pip into installing a drastically inconsistent package set because two things they use in proximity to each other have radically different assumptions. That set may work for them and they may not know what they did to produce the "working" situation. Setting the proxy to one of these full ecosystem channels will have a different package index and although most of the time the solver will notice that it can't find a workable solution to some updates and fail, some packages don't completely specify (especially) their upper bounds so they might be pulled in where they won't work. Especially if this updates one of the "working" packages to one that doesn't have the same API as the one that was updated, that person might be really upset and have a difficult time replicating the situation they wanted.

- ***What's this about "We can't lock down absolutely everything but there are things we can do to help." -- if there's a sandbox running based on seccomp... what can't be done? Would love an enumeration of what's seen as possible, and what's left as "impossible".***

It's not seccomp that can't do what we want, it's that setuptools and distutils are live python code, people can copy old versions into their projects, emulate them, work around issues by patching them. People have specific needs in builds that won't necessarily be amenable to a locked down build environment and need exceptions because they can't fix them in short order.

There are a lot of reasons for a fully locked down sandbox not to be able to exercise what one might think of as complete control, even if it can protect every file on disk from being modified when it has a hash that should be known.

What I envision is strictly checking (and failure if strict checks fail) of the installed files during and after package set updates, as well as an opt-out prohibition on modifying files whose hashes are (or should be) known. We will infiltrate processes downstream of the sandbox (unless running in an isolated environment such as docker) and make an effort to ensure that the python package set is pulled and updated (and manipulated) according to consistent methods, but as above, people can and do make their own changes for various reasons and may not be able to stop doing.

- ***I'm getting slightly mixed signals about who is involved, fwiw :) Maybe this has been more discusses elsewhere or offline, but from this doc above: I see both several specific names, but also a description of roles that need to be filled. Are the names already listed... People who have seens this design doc? Are interested and contributed revisions or reviews to the proposal? Will be continuingly involved in review or as downstream users? Will work directly on the project, or, will be working in parallel directions regardless of this grant? Are the other company names involved, or? Friends and additional eyeballs are always good, but I'm curious what degree of interest/alignment/commitment are being expressed :)***

Muinin itself is a small startup looking to break into software security and software supply chain, with the people you've seen mentioned. Its membership is largely business focused with the exception of myself (art yerkes, now in an advistory capacity) and jim gettys. Our plan from the beginning of this proposal was to use the funds from the proposal to hire and pay for developer and devops time and pay the business for managing these. I think with proper guidance and a bit of elbow grease there's nothing stopping things from being.

Proposal from Muinin Corp. pbc, which will manage and have responsibility for the project, if awarded.

Project manager: Ramesh Rao

Architect/Advisor: Art Yerkes

Lead Coder/RB: Vagrant Cascadian (likely)

Major Coder: S Sairam

Python Coder: TBD

Other Support: John Ryan, Dan Conway, Jim Gettys

Thanks in advance for any more clarifications you can share :) I really am deeply excited by all work in this space!

Appreciate your questions and enthusiasm 🙂
