Git Contributor's Summit, 2022

September 14, 2022 – Chicago, IL, USA

Bundle URIs

- Unlike packfile URIs, includes refs, does not need to be delta-ed against what server sends
- Doc checked into Documentation/technical
- URI can be provided by user at CLI or advertised by server
- Most users won't experience anything if they git-clone, but it will only benefit the git hosting providers. It will allow them to offload data to CDNs, being closer to the client.
- With bundle files you can download them and start of from there and fetch the objects you're missing in a regular manner.
- Jrnieder: Packfile URIs and Bundle URIs are trying to achieve the same thing. How can we duplicate efforts? E.g. how can we prevent the client from leaking information to a possibly untrusted server?
- Stolee: Are you want to provide a way to provide authentication?
- Jrnieder: Analogy to the web - you don't want to leak information to websites you don't trust. The security model is pretty complicated, we don't want to replicate things like same origin policies.
- Stolee: So, e.g. the server provides a hash of the content expected at the bundle URI and the client can verify? We wanted to explicitly avoid that because we don't want the server and bundle provider to need to know anything about each other.
- Jrnieder: Compare to packfile URIS - Packfile URIs are only advertised for the server, so the security model is mostly the same as a "regular" fetch/clone
- Jonathantanmy: Another difference: the objects in bundles must be associated with refs, you can't just have e.g. large objects. Packfiles can contain arbitrary objects.
- Stolee: Let's talk about the security model more on the mailing list
- Ævar: We're also open for a breakout session on this topic

SHA-256 transition

Notetaker: ttaylorr (thanks!)

- (brian) Functional version of "state four" implementation with only SHA-256 in the repository
- Interop work (to use sha1 and sha256) is mostly stalled, brian is mostly not working on it at the moment
- Current implementation is partially functional, though failing a lot of tests. Can write SHA-256 objects into the repo, according to the transition, will write a loose mapping between SHA-1 and SHA-256, along with index v3 with the hashes for both
- When you index a pack, computes both hashes and stores them in the loose object store or pack

- Tricky part is when you're indexing a pack, you don't always get all blobs before all trees, before all commits, etc.
- In order to rewrite a commit from SHA-256 -> SHA-1, you need all reachable objects before in order to compute the hash. Try to look up in a temporary lookup table ahead of time, and lazily hash the object we're going to get and come back to it later.
- "Rewind the pack" to compute the proper objects, which works
- For submodules (currently unwritten), going to send both hashes over the wire, but unfortunately no way to validate those in real time. If your submodules are checked out, rewritten automatically.
- brian working on it slowly as they get to it, hopes that their employer will devote more time to it
- Wants to also work on libgit2 at the same time, since it doesn't yet understand SHA-256, though they hope that somebody else will work on it, since they are tired of writing SEGVs :-).
- (demetr): what if you have a remote that speaks only SHA-1?
    - Goal is to have that information come over the pipe, and rewrite into SHA-256 upon entering the new objects into the repository
- (demetr): can you then push a converted-into-SHA-256 repository back to a SHA-1 repo
    - Goal is to be able to do that, unless you have a SHA-1 collision, in which case it won't work.
    - No major hosting platform yet supports only SHA-256 repositories, though maybe Gitolite and CGit do
- (Peff): so, in the worst case, index-pack takes twice as long?
    - brian: depends on how many are blob objects, since only takes a single pass
    - Will try to rewrite objects in as few passes as possible
    - May need multiple passes in order to visit objects in topological order
    - Actually: worst case is N where N is the maximum tree depth
- (Stolee): what you really need is reverse-topo order on the object graph
    - brian: yes, would be nice if the server sent them in that order. But the server doesn't know how to do that.
- (Emily): so for something like shallow/partial-clone, the server needs to be able to do SHA-256 for you to compute it yourself?
    - brian: there will be a capability, since data needs to come over the pipe for submodules, and could be extended for shallow and partial clones as well. Would fit into protocol v2, and will be essential for submodules, so will have to exist regardless.
    - Hopefully server has that information, though how that expensive will be to compute is highly dependent.
- (jrn): submodules have to be updated, do you have an idea of what that protocol change will look like?
    - brian: fuzzy idea, but nothing concrete yet
    - (jrn): this reminds me of the early days of partial clones where we talked about "promised" objects at the edge and associated metadata

- (Toon): so no interop, but is there a way to do a single step conversion from SHA-1 to SHA-256?
    - brian: yes, you can use fast-export and fast-import. Currently any signatures references are broken, but in the future would like to update them (that code exists, but it hasn't been upstreamed)
    - doesn't quite work with smoothly submodules, since you have to rewrite them first, then generate a set of marks, and then export and import
    - verified with git/git, resulting index isn't substantially larger (basically 32 bytes per object, along with slightly larger commit and tree objects)
- (demetr): Could be significantly larger if you have a zillion commits
    - brian: we'd have other problems before then :-).
- (Elijah): common in commit messages to refer back to earlier commits. Do we want to rewrite those?
    - brian: maybe, depends on future plans if/when we deprecate earlier hash algos
    - (jrn): Don't have a good way to retroactively change commit messages, but we do have git notes. First instinct is to use notes for this kind of historical reference info
    - (Terry): annotated tags?
    - (Elijah): filter-repo does this kind of commit message munging

Merge ORT

Notetaker: vdye (thanks!)

- Git has multiple merge backends - default was "recursive", now "merge-ort"
- When merge-ort was written, it was intended to be a replacement
- Would people be okay removing the old merge-recursive code? ORT was meant as a drop-in replacement, but it does have some differences in behavior
    - If it is okay, what's the timeline?
- (Taylor): gradual guarded by config might be a good approach (e.g., "merge.recursiveIsORT")
- (Johannes): describing the differences in behavior
    - Merge-ort rename detection is always on, but merge-recursive is opt-in
    - ORT uses histogram diffs (arguably more correct, matching unique lines in diffs). Leads to (sometimes) merge conflicts where merge-recursive didn't find it, but ort did. Still probably more correct, though. Much more rarely, it's the opposite - recursive found a conflict, ORT did not
    - Conclusion: wait 2 major versions to deprecate
- (Taylor): maybe we should add "turn off rename detection" option
- (Johannes): should we even give users that option in the first place? If they don't have it, they won't be as upset when we get rid of it ;)
- (Peff) how long has ort been the default? 2 versions. Now people have recursive as a escape hatch. But we don't know if/when people use if. Also recursive with find-renames is an escape hatch.

- (Emily): we know how often people are using escape hatches (at Google), could take the same approach with this option
- (jrn): do we have other signals for how often this escape hatch is used? Stack Overflow posts?
    - No one's named it as a solution on the mailing list, though, so we don't know from that medium
    - Agree with Johannes, might be best to not give the option to users because this way we have more chance of getting signal.
- (Peff): the mailing list isn't representative of the larger Git community, so people bringing it up to the mailing list might not be indicative of usage
    - Leaving the hatch doesn't seem like it'd incur a huge maintenance burden
- (Terry): some distros might have significant propagation delay, should probably bake in extra time because of slow adoption
- (Ævar): I'm happy to follow your decision
    - Some behavior difference, but it's working *better*
    - At some point, we should be willing to say "if you need an old feature, use an old version"
- Some observed differences might be libgit2 recursive vs. ORT, unclear which ones though
- (Johannes): We can bake in a deprecation notice, like: if you see something wrong, now is the time to bring it up! We'd rather fix it now
- (Jeff): most users won't be bothered - they'll see a conflict and resolve it, without thinking about which algorithm generated the conflict
- (jrn): for most usage, this is a completely safe change. The discussion comes up because of the fear that some user might have some use of merge they regularly do that hangs, that kind of thing, rather than the subtle merge resolution changes that we've discussed. I think we're safe. :)
- (brian): observing something like Debian or other LTS versions of projects - if there isn't a lot of screaming after a couple months, it's probably safe. Even if you wait a decade, though, there'll always be one or two who suddenly encounter issues with the new feature after the old is long deprecated.
- (Ævar): haven't seen any huge warning in the merge docs saying we've changed something, but with how many users are already using it, it's likely very few people will ever notice

git clone --filter=commit:0 (jonathantanmy)

- Partial clone that can omit commits (we already support trees and blobs)
- Pros:
    - Don't need all commits, save network and disk I/O. There's a repo at Google that grows so quickly that having just commits is too much
- Cons
    - Git assumes that all of the commits are present locally; very pervasive assumption.

        - Blobs don't have outlinks, not a problem. Tree depth is somewhat limited. Commits go all the back to the beginning of the repo
        - `git bisect` without commits
        - Lose out on optimizations like `fetch` skipping negotiator, commit graph generation numbers
- Has everyone else thought about this?
- Peff: Compare to shallow clone (create a 'graft' and pretend that the commit has no more parents). How do we handle the continuous N + 1 fetching? Jonathantanmy: Not a big issue, we can batch fetch. It's jumping around that's a problem
- Peff: What if the server sends the commit graph?
    - Taylor: we could just send the generation number(s) of the parents of the commits on the boundary of what you're sending.
    - Emily: We can't verify it though, we'd have to just trust the server
    - Taylor: true, but that's the case even if you send the whole commit graph, too
- Jrn: Partial clone - we know the server is there, so we still have a "full clone", but part of the "full clone" lives in the server. There are git services that don't need a full copy of the repo, e.g. for CI, we only need a view of the directories we're building.
- Two use cases for partial clone
    - Shallow clone replacement: user only cares about a single commit
    - Operation that involves history walking (e.g. git describe). We might as well fetch all the commits (i.e., convert to tree or blob filtering when we notice this operation). Are these operations distinguishable?
- Rdamazio: What if all of the history walking happens only on the server? (e.g. git blame). Jrnieder: For git blame specifically, that makes sense, but are you thinking of other things like git log? Rodrigo: Yes
- Johannes: That doesn't sound like it will scale. Stolee: At GitHub, we already do run blame on the server Rodrigo: At google, we precompute that
- Terry: More and more things want "stateless" operations (don't care about history) - that's probably the *majority* of use cases. There's also a popular use case of "1 week/month" of history. It would be great to not pay the penalty of fetching all commits. Today, we only have shallow clone, which pretends that history is different from what it actually is, and it's very difficult to maintain this on the server (sending not enough objects, sending too many objects). But filters are much easier to maintain.
- Victoria: Is this a replacement for shallow clones then? Terry: Yes.
- Stolee: 2 technical areas of apprehension
    - VFS for git tried to do this by having only the initial commit and fetching later objects one-by-one. Didn't work at all, was very slow.
    - Treeless clones - when traversing the tree, we keep refetching the tree when we traverse it.
- We would have to drastically rework how GIt interacts with partial clones
- Taylor: Or we could teach the server to preempt the operations ("i'm going to run git log, send me the right things")
    - Stolee: Or run it on the server
    - Taylor: yes, that would be the other extreme approach to this

- Jrnieder: With treeless clones, we don't propagate the filter on the catch up fetch, and there are some code locations that assume that if we have a tree, we have all of its children. If anything, commit filters are even easier because we have nothing - so we can do "all or nothing"
    - Stolee: I agree that it's simpler, but I still think it'll be really slow. So either, we need to do something much smarter than object-by-object fetches, or to prevent users from running problematic commands. We would eventually have to fix the problem for treeless clones, so what if we start with full commit history, but not all of the trees. We can fix that first before starting on commit filters
    - Jrnieder: I can see the need for all of the commits up until a certain point in time, but I don't know if there's a need to solve the *general* problem of omitting arbitrary commits e.g. jumping around in bisect
- Rodrigo: We have some experience for doing this with Mercurial at Google - we hide the full history, users know they exist, but they can refetch if they wish. Stolee/Peff: That sounds like reimplementing shallow clone.
- Taylor: Is there any other kind of filter other than commit:0? Jonathantanmy: No plans yet.
- Peff: Wouldn't you need to implement the general case to do batching of commits in "git log"? Jonathantanmy: Maybe not, we could e.g. reuse the shallow clone protocol.

Managing ever growing pack index sizes on servers

- Some repositories have over 15 years of history with 1000 active developers, so pack indices can be between 1 and 2 GB. "GC pack" contains everything reachable from refs/heads/* and refs/tags/*
- Time-based slicing for repositories to allow smaller repositories. "Remove" history from before a certain point. Done by taking a shallow clone and using that as the new repository.
- What about folks who are only interested in the last week's history?
- Pack repositories based on time-based slicing. Moving back to older history can fall back to older packs as necessary.
- Some people, like documentation folks, don't need the entire history and might be fine with a more limited environment.
- Chromium packs to three packs: one is a cruft/garbage pack, and the other are reachable objects. refs/heads is packed into one pack, and refs/changes (PR-equivalent) are in the other.
- JGit doesn't have a reverse index *yet*
- Taylor: considering packing reverse index into main index. The tension is that we need to make using multiple packs more flexible. Introduce bitmap chains when repacking to make things more stable and less expensive.
- Stolee: Consider older packs that are stable and only repack newer things.
- Peff: One reason not to have lots of packs on disk is missing out on deltas. We could use thin packs on disk.
- Stolee: Future goal is to only include full delta chains in the stable packs.

- git gc –aggressive used to make really deep deltas and has been fixed to be less aggressive to avoid runtime performance costs. Between 10 and 50 shows real performance improvements but the old default was like 200.
- The original numbers were picked randomly without measurement.
- Patrick: GitLab maintenance architecture is evolving. Each push is incremental (repack into one pack) or full repack (everything into one pack with deltas).
- Stable ordering for determining preferred objects (SHA ordering is not suitable).

Server side merges and rebases (& new rebase/cherry-pick UI?)

Notetaker: chooglen (thanks!)

- Elijah: tried to implement the git side of the cherry pick as flags to git merge subcommand, but everything turned out to be incompatible. Used git merge-tree instead, much better, but this doesn't create a new commit, only a new top-level tree.
- Rebase and cherry-pick is even more tricky because we need sequences of commits. Does the current UI make sense?
- I want to create commits on a not-checked out branch, or rebase, or cherry-pick. Not only on the server, but on any client.
- Rebase skips cherry-picks, but that is probably just an optimization for when rebase for a shell script. Always doing cherry-picks is faster these days, but is a behavior change
- Creating a new commit and modifying the working tree - this lets hooks run, but I don't want them to run the server
- Rebase and cherry-pick are typically centered around HEAD, I would prefer to replace with just a commit range. If you don't make the assumption around HEAD, cherry-pick and rebase aren't that different.
- How do we display conflicts generated on the server side so that ? We don't have a representation for that. Taylor: Probably just block the operation on the server. Elijah: That's my intuition too.
- We have a lot of users who want to cherry-pick a commit on a bunch of LTS branches, it would be great if they don't have to check out those branches.
- What about cherry-picking to older branches? It's super slow to check out the old branch and it's a big pain to update.
- Want to be able to replay merges. Not just like rebase --rebase-merges, but with extra content/resolutions
- Emily: Rebase has famously bad UX. Could we create a new command that fixes the problems, like checkout and switch? Elijah: I'm worried that I'll copy the old terminology, so I'd need feedback on that.
- Stolee: We could rework the underlying API that supports rebase and cherry-pick and use that for the new UX.
    - Jrnieder: We don't have plumbing commands for this yet, which would be very nice to have. For changes motivated by "cherry-pick has this bad behavior", if we're not making an overall better UX then I'd encourage "go ahead and make cherry-pick no longer have that bad behavior"

    - Jonathantanmy: I think base + theirs + ours is good enough. Elijah: Sounds like git merge-tree, I don't think that's enough for the server case. I'm sometimes porting over multiple commits instead of just one, ort can do some optimizations on that, but one-by-one invocations would lose that info. Also, this isn't enough to replay merges.
- Peff: It would be good to have a machine-readable representation of a conflict that the server can serve, but also can be materialized by client tools. Taylor: It would be even cooler if we could push that representation and have "collaborative" merge resolution. Elijah: Merge-tree can output files with conflict markers. We'd have to add info to represent the index conflict. With rebase, we'd need to represent different conflicts at different points.
- Martin: Does ort handle conflicts with renames? E.g. renaming two files to the same name. Elijah: Yes
- Elijah: One format would be input to git update-ref --stdin, so instead of making all of changes, you could output the data that git update-refs can ingest later.
- Waleed: Do you support rebasing non-linear sequences? Elijah: Yes, but.. (didn't hear)

State of sparsity developments and future plans

- (Victoria) Integrating commands with sparse index, making them compatible with sparse indexes, not un-sparsifying the index before executing themselves
- Have worked on some more recently
- GSoC student has worked on a handful as well
- Near-term future is about finding commands that need to touch the index, don't support sparse index, and then make them compatible
- In some cases, that is going to require expanding the index to be non-sparse, especially if you are touching something outside of the sparsity cone
    - That is somewhat straightforward
- More interesting questions: what is the future of sparsity? Recently, Elijah pushed a change to make sparse-checkout's cone mode the default (nice, since it is required by sparse index)
- As we move forward, what should we change the defaults of?
    - Sparse index for sparse checkouts in cone mode?
- Scalar as a testing ground for larger features, including sparse index
    - Could make sparse index the default in Scalar for cone-mode sparse checkouts, and then see how it goes
    - Or, could just go for it sooner (after we have integrated sparse index with enough commands)
- A handful of internal, logistical things that would have to happen for sparse index to become the default. Currently, commands are assumed to not work with the sparse index.
- Question for everybody: what is a good balance between pushing sparse index, and waiting to introduce it to more users by holding off on changing the default.

- (Stolee): sparse checkout and submodules became a difficulty when mentoring their GSoC student.
- (JTan): possible to decouple sparse index and cone-mode sparse checkouts from each other? This would be easy to test - turn it on, all of the test suite automatically uses it. Jrnieder: This sounds ok for the filesystem, but I don't know how this would work for this "VFS-backed Git" idea on the spreadsheet. (other things…)
- (Stolee): We need cone mode today because they're the only way to definitively say that we've reached the boundary. But we can also expand the idea of "cone" to allow more paths (files instead of directories) in the cone.
- (Taylor): What do we need to tell subcommands to assume that sparse index is supported? (Victoria) Gut feeling for the most part. (Jrnieder): I'd prefer this to happen sooner rather than later. This is easier for maintainability since we don't have to worry about commands being in two possible modes of operation. We can break these incompatible APIs by renaming them to prevent them from being misused by new commands.
- (Victoria): So just break things that always use the full index? Sounds ok. (Stolee) This sounds similar to the_index macros, which we've tried to remove for the most part but we've stopped. Doing this conversion everywhere sounds extremely difficult - we've done an audit on this. (Jrnieder): Oh, I just meant renaming the API without changing semantics. Intentionally break everything.
- (Victoria): We'd need to write new tests for lots of commands because the existing tests don't actually interact with the "sparse" parts of the index.
- (Ævar): Is this just a matter of telling `git init` to initialize a sparse index? (Stolee): No, we need to force the tests to work on sparse directory entries.
- (Jrnieder): This sounds like a good fit for feature.experimental
- (VIctoria): Is sparse index a good git default instead of just "for large repos"? I'd think yes. (Jrnieder) Yes I think any sparse checkout user would want this. (Stolee): LIterally every command that touches the index has been converted (used for Microsoft Office monorepo), so it's just a matter of doing this for the whole project.
- (Elijah): I would like partial filters from sparse patterns. --filter=blob:none doesn't let you disconnect from the server.
    - (Jrnieder) The DX of sparse checkout + blob:none has been pretty good. (Elijah): but you need to stay connected to the server. (Jrnieder) Ah, thanks for explaining, sorry for the confusion (Elijah) It would be great to have "sparse clone"s and have commands that work just inside of that cone when disconnected. Make "grep", "log", etc respect the sparse pattern
    - (Stolee): We've thought about this, but it is very expensive on the server side and makes bitmaps unusable. Alternatively, we could start with blob:none and then backfill. That sounds more promising, but that's not just a plain partial clone.
    - (Jonathantanmy): FYI there's a protocol feature that already allows clones to specify a sparse filter (referencing a blob with sparse patterns that's present on the server), but I don't know of any implementation that has this enabled.

    - (Jrnieder): Can we delete this? (Github folks): We don't like it, we invented the uploadpackfilter config to disable it :) Is this just cleanup? (Jrnieder): Yes, and this dead end will stop being a distraction.
    - (Peff) We could already implement most of the backfilling using current commands, but that might skip over some delta-ing optimizations. We could have a protocol change to provide the path as a hint to the server.

Ideas for speeding up object connectivity checks in git-receive-pack

- Patrick: At GitLab we've noticed some repositories have become slow with a lot of references.
- Initially I thought it was the object negotiation.
- The connectivity checks seem to be the culprit. The connectivity check is implemented quite naively. "git rev-list <commits> --not --all"
- A while ago I tried to optimize it. Stop walking when you reach an existing object. List feedback was that I had not gotten the semantics right, since that existing object is not necessarily pointed to by a ref and not everything it references is necessarily present in the repository.
- Trial 2: optimize rev-list. Sped up connectivity check by 60-70%! But the motivating repositories had grown, so it was still too slow.
- How do other people do it, with millions of references?
- Terry: Yes, we've seen this problem.
- Just the setup time for the revision walk takes a long time. Initially we weren't using reachability bitmaps; using those also helped. Then using subsets of references — first HEAD, then refs/heads/*, …
- Jrnieder: Initially we did this for reachability checks; do connectivity checks do it, too? Terry: Yes, Demetr did that.
- Demetr: When the connectivity check fails, we fall back to a fuller set of refs.
- Patrick: Many of our references are not even visible to the public. If Git makes configurable which refs are part of the connectivity check, that would already make things faster in our case.
- Taylor: Did you experiment with reachability bitmaps? Am I remembering correctly that they made things slower?
- Patrick: We did some experiments with reachability bitmaps, but not specifically for this problem. In those experiments they did make some things slower.
    - Taylor: one thing that you could do is make the bitmap traversal by building up a complete bitmap of the boundary between haves and wants instead of a bitmap of all of the haves. Involves far less object traversal, and there are some clever ways to do this.
    - Taylor: As long as you can quickly determine the boundary between the haves and wants for a given request, the connectivity check should be as fast as you need.

- Peff: One difference between how Git and JGit does this is JGit is building up a structure of “what is reachable”. You could persist a bitmap representing “here’s everything that’s reachable from this repository” and subtract that out; that would help with many cases. The problem with one reachability bitmap like that is that it goes stale whenever someone pushes something. But you could make a set of “pseudo-merge bitmaps” for each group of 10,000 refs or so. Especially if you’re clever about which refs you do that for, that can be a significant improvement (“these 2,000,000 refs haven’t been touched for a year, so I can use this bitmap and don’t even have to examine them”).
- Terry: There are two ways that unreachable objects appear. One is by branches being deleted or rewound. The other is failed pushes where objects were persisted and the ref update didn’t succeed. I wanted to distinguish between objects that are “applied” and “unapplied”, became very thorny. And with that, on a branch rewind, we can calculate what just became unreachable.
- Waleed Khan: Object negotiation with bloom filters paper
    - Kleppman and Howard 2020
    - Blog post https://martin.kleppmann.com/2020/12/02/bloom-filter-hash-graph-sync.html
    - Paper https://arxiv.org/abs/2012.00472
    - Quote: “Unlike the skipping algorithm used by Git, our algorithm never unnecessarily sends any commits that the other side already has, and the Bloom filters are very compact, even for large commit histories.”

Alternative ways to write to VFS-backed worktrees (e.g. write($HASH) instead of write(<bytes>))

- Google tried to use the sparse-index to control what gets materialized by in a VFS-like system. What if we replaced the write() syscall + writing bytes and write a hash instead. It could save a lot, e.g. in an online IDE.
- Stolee: The “write” analog of FSMonitor for VFS-for-Git is the post-index-change hook. We suppress the writes by manipulating the index and then communicating the write back to the VFS.
- Jrnieder: This is bigger than just the “write()” call; if we have a git-aware filesystem, we can be much less wasteful than we are today. E.g. FSMonitor can only make `git status` so fast because we still have to stat(), but with a VFS, we could ask the VFS what has changed.
- Do we think this is useful for anything other than a VFS? Do we still want this even if it’s only useful for VFS?
- Stolee: We could make FSMonitor more git-aware e.g. it doesn’t know about writes that we make. JeffH: We also can’t say “ignore .o files, i only care about source files”. That also helps greatly. Writing a hash instead of the contents is probably about as expensive, most of the savings are in avoiding the stat() calls. This also sounds racy.
- Emily: Another reason to do this work is that this is a good jumping off point to libify the Git internals. Is there any reason not to do that? Jrnieder: To make this concrete: do you mean for example creating a worktree.h with a vtable of worktree operations and having

things talk to that instead of the FS? Emily: Yes. Things like that are the reason why we have libgit2, so what if Git could just ship its own library. Bmc: Libgit2 is used in lots of places like the Rust package manager (Cargo). The problem is that Git is GPLv2, which is not usable by lots of folks.
- Stolee: The stance of the Git project is that the API is the CLI, not individual files. But I think this is a good thing to have for the project as a whole, even just internally.
- We can finally have unit tests!!??!

How to run git securely in shared services

- Kevin: What do we think about GIt on a shared device? e.g. Git trusts the repo more than the global config, but the repo might not be trustable. What do we think of, say, inverting this precedence?
- bmc: Repo config overriding global config is an important feature we should not lose. But I could imagine some global option that affects that behavior — making it very explicit, on that particular machine.
- Stolee: I wrote an email to the mailing list months ago about this subject. Title “What is Git’s security boundary?” Concrete proposal: anything that executes an executable could go through a hook that is installed at the global or system level, and local config can refer to that. E.g. “please run vim, which has been controlled at the system level”.
    - I got zero traction, but there’s some prior art.
- Taylor: We think of Git as special in this way. For “make”, we wouldn’t ask this same question.
- Jrnieder: With “make”, it’s very clear to users that arbitrary commands might be run, but users don’t have that same expectation when just browsing code.
- Taylor: We could create a mode that ignores repo config, and that makes prompts safer. But we’re inherently make-like.
- Jrnieder: That’s obvious to us, but not to most users. I think we’re quite far from having Git’s security model match users’ mental model and I think it’s hard to change the behavior but would be even harder to change users’ mental model in this example.
- Jrnieder. I wish we could separate out “repo properties” from “actual config”: keep my user preferences separate from the things that Git needs to run.
- Ævar: Emacs has solved this. Emacs can run arbitrary code for all kinds of things, but it prompts users to approve the code first. We could allowlist harmless config, and then only prompt users for sketchy things. Taylor: This kind of allowlisting sounds impossible though. Bmc: Can we do this just for core.repositoryformat and extensions.*?
- (much spirited discussion, did not hear)
- JTan: If we want repositories to still be movable, how would we maintain this allowlist? Ævar: There are ways to do this for just certain variables, or certain variables in certain paths, etc. We have a lot of space to do the right thing.
- Bmc: It’s important to make this behavior configurable from the command line.
- Ævar: I was experimenting with this because I wanted a way to have config in-repo. It would be very useful even if we could only control a subset of config

- Jrnieder: We already have some defense against hooks-and-config in special cases e.g. uploadpack doesn't trust the local repo's hooks. Suppose we have completely solved the problem of protecting against those; are we comfortable with changing the threat model to encompass normal commands in local repositories?
- Peff: I think the safest option is to just ignore the in-repo config altogether. Johannes: But the unsafe thing isn't parsing the repo, it's executing code. We could just shift the boundary to "don't execute code outside of safe.directory".
-

TIme's up!