1 Summary

Implement a prototype job worker library that provides an API to run arbitrary Linux processes.

2 Rationale

This exercise has two goals:

- It helps us to understand what to expect from you as a developer, how you production code, how you reason about API design and how you communicate when trying to understand a problem before you solve it.
- It helps you get a feel for what it would be like to work at Teleport, as this exercise aims to simulate our day-as-usual and expose you to the type of work we're doing here.

We believe this technique is not only better, but also is more fun compared to whiteboard/quiz interviews so common in the industry. It's not without the downsides - it could take longer than traditional interviews.

Some of the best teams use coding challenges.

We appreciate your time and are looking forward to hack on this project together.

3 Requirements

Start with a very brief Google doc that covers the edge cases and design approach and post it to the Slack channel. After the doc is approved, implement interfaces and an example program using the library.

Add a couple of high quality tests that cover happy and unhappy scenarios.

Split the submission into 1-2 pull requests for us to review. We will review every pull request and provide our feedback.

We are going to compile the program, test it and get back to you.

Library

> Worker library with methods to start/stop/query status and get an output of a running job.

4 Guidance

4.1 Interview process

The interview team will join the Slack channel. The team consists of the engineers who will be working with you. Ask them about the engineering culture,

work and life balance, or anything else that you would like to learn about Teleport.

Before writing the actual code, create a brief design document in Google Docs or markdown in Github and share with the team.

This document should consist of key trade-offs and key design approaches. Please avoid writing an overly detailed design document. Use this document to make sure the team could provide feedback on your design and demonstrate that you've investigated the problem space.

Split your code submission using pull requests and give the team an opportunity to review the PRs. A good "rule of thumb" to follow is that the final PR submission is a formality adding a small feature set - it means that the team had an opportunity to contribute the feedback during multiple well defined stages of your work.

Our team will do their best to provide a high quality review of the submitted pull requests in a reasonable time frame. You are spending your time on this, we are going to contribute our time too.

After the final submission, the interview team will assemble and vote using a "+1, -2" anonymous voting system: +1 is submitted whenever a team member accepts the submission, -2 otherwise.

4.2 Code and project ownership

This is a test challenge and we have no intent of using the code you've submitted in production. This is your work, and you are free to do whatever you feel is reasonable with it. In the scenario when you don't pass, you can open source it with any license and use it as a portfolio project.

4.3 Areas of focus

Teleport focuses on networking, infrastructure and security.

These are the areas we will be evaluating in the submission:

- Use consistent coding style. We follow Golang Coding Style for the Go language. If you are going to use a different language, please pick coding style guidelines and let us know what they are.
- Create one test for unhappy scenario.
- Make sure builds are reproducible. Pick any vendoring/packaging system that will allow us to get consistent build results.
- Ensure error handling and error reporting is consistent. The system should report clear errors and not crash under non-critical conditions.
- Avoid concurrency and networking errors. Most of the issues we've seen in production are related to data races, networking error handling or goroutine leaks. We will be looking for those errors in your code.

4.4 Trade-offs

Write as little code as possible, otherwise this task will consume too much time and code quality will suffer.

Please cut corners, for example configuration tends to take a lot of time, and is not important for this task.

Use hardcoded values as much as possible and simply add TODO items showing your thinking, for example:

Listing 1: TODO example

```
// TODO: Add configuration system.
// Consider using CLI library to support both
// environment variables and reasonable default values,
// for example https://github.com/alecthomas/kingpin
```

Comments like this one are really helpful to us. They save yourself a lot of time and demonstrate that you've spent time thinking about this problem and provide a clear path to a solution.

Consider making other reasonable trade-offs. Make sure you communicate them to the interview team.

Here are some other trade-offs that will help you to spend less time on the task:

Do not implement a system that scales or is highly performing. Describe which performance improvements you would add in the future. High availability. It is OK if the system is not highly available. Write down how you would make the system highly available and why your system is not. Do not try to achieve full test coverage. This will take too long. Take key component and implement one or two test cases that demonstrate your approach to testing.

4.5 Pitfalls and Gotchas

To help you out, we've composed a list of things that previously resulted in a no-pass from the interview team:

- Scope creep. Candidates have tried to implement too much and ran out of time and energy. To avoid this pitfall, use the simplest solution that will work. Avoid writing too much code. We've seen candidates' code introducing caching and making many mistakes in the caching layer validation logic. Not having caching would have solved this problem.
- Data races. We will scan the code with a race detector and do our best to find data races in the code. Avoid global state as much as possible; if using global state, write down a good description why it is necessary and protect it against data races.

- Deadlocks. When using mutexes, channels or any other synchronization primitives, make sure the system won't deadlock. We've seen candidates' code holding a mutex and making a network call without timeouts in place. Be extra careful with networking and sync primitives.
- Unstructured code. We've seen candidates leaving commented chunks of code, having one large file with all the code, not having code structure at all.
- Not communicating. Some candidates have submitted all their code to the master branch without raising pull requests, which does not give us the ability to provide feedback on the various implementation phases. We are a distributed team, so structured, asynchronous communication is critical to us.
- Implementing custom security algorithms/authentication schemes is always a bad idea unless you are a trained security researcher/engineer. It is definitely a bad idea for this task - try to stick to industry proven security methods as much as possible.

4.6 Scoring

We want to be as transparent as possible on how we will be scoring your submission. The following table provides a description of different areas you will be evaluated on and how they will affect your overall score.

Description	Points Awarded	Points Subtracted
The submitted code has a clear and modular structure	+1	-1
The candidate communicated their progress during the interview	+1	-1
The program builds are reproducible	+1	-1
README provides clear instructions	+1	-1
The candidate outlined the key design points in the design document	+1	-1
The code has no obvious data races and deadlocks	+1	-1
The code provides examples of tests covering key components	+1	-1
The code provides clear error handling and reporting	+1	-1
The program is working according to the specification	+1	-1
The candidate demonstrates ability to handle and apply feedback	+1	-1

4.7 Scoring

It is OK to ask the interview team questions. Some folks stay away from asking questions to avoid appearing less experienced, so we provide examples of

questions to ask and questions we expect candidates to figure out on their own.

Here is a great question to ask:

"Is it OK to discard the output of a process that exceeds a predefined hard-coded value?."

It demonstrates that you thought about this problem domain, recognize the trade off and are saving you and the team time by not implementing it.

This is the question we expect candidates to figure out on their own:

"What version of Go should I use? What dependency manager should I use?"

Unless specified in the requirements, pick the solution that works best for you.

5 Tools

This task should be implemented in Go, C++, Rust or Java and should work on 64-bit Linux machines.

6 Timing

It should take you from 4 to 24 full hours to complete the challenge.

You can split coding over a couple of weekdays or weekends and find time to ask questions and receive feedback.

Once you join the Slack channel, you have a maximum of 1 week to complete the challenge.

Within this timeframe, we don't give higher scores to challenges submitted more quickly. We only evaluate the quality of the submission.

We only start the coding challenge if there are several open positions available and let all candidates finish the code submission.