Cloudflare Zero Trust Client GUI

Take home test instructions

Version	Date
1.0	11th August 2023

Cloudflare ZT Client GUI

Cloudflare ZT Client GUI

Introduction

Thank you for your interest in joining Cloudflare, and applying for the role of Zero Trust Client GUI engineer.

Our team greatly values low-complexity, highly-documented, thoroughly-tested code, and we’re excited to see how you write, architecture, and test your code.

To understand your skills in those areas, we have opted for a take home test to give you more time to achieve something you might be proud of, and to remove the stress that face-to-face interviews can bring.

We have designed this interview to closely match the type of work that we do in our team.

We understand that not everyone will have the same amount of time available to do this task, and that is OK. Just like at work, we do not always have the time available to craft features the way we’d like to, and have to make trade-offs.

When you’re done, our team will review the code, and you’ll meet with a member of the team to chat about the decisions and trade-offs you’ve made.

If you have any questions, issues, or concerns during this exercise, please reach out to your recruiter as soon as possible

Cloudflare ZT Client GUI

Take home test

Background

At a high level, this task mimics our VPN product. Our networking stack is implemented by a networking daemon/service, and the GUI app interacts with it. For example, when the user clicks the connect button, the app sends a request to the daemon/service. The daemon/service then tries to establish a connection to the VPN server, and communicates the response with the GUI, which then reflects the status (e.g. connected or error).

Download

https://drive.google.com/drive/folders/10z22krSrFcyApr4gsrT-bsg3dKoQQsuJ?usp=sharing

The download link contains this document, as well as the mock daemon you will need for this task. It is given pre-compiled, along with its source code should you wish to compile it yourself. To compile it, you will need Rust and Cargo.

This document contains instructions on how to interact with the mock daemon.

Cloudflare ZT Client GUI

Instructions

You are asked to create a simplified GUI application that represents a VPN app. The GUI application will have to interact with a mock registration API and the provided mock networking daemon (macOS/Linux) / service (Windows). This document contains instructions on how to interact with both.

In this document, *"the app"* refers to the GUI application, and *"the daemon"* refers to the networking daemon/service.

The app should offer a way for the user to connect/disconnect from the VPN (implemented/represented by the daemon), and show the status of the connection. Which states are represented and how they are represented is left as an exercise to the candidate. We are interested to understand the decisions you make and why.

The app should be usable, responsive, intuitive, and crash-free. It must also handle errors appropriately. How the app manages/responds to errors is again left as an exercise to the candidate.

App flow

The app and the daemon both start in the disconnected state.

When the app starts, it should check the status of the daemon.

To connect the daemon, the app must first contact the registration API to retrieve an authentication token. The app then sends a `connect` request to the daemon, with the authentication token as a parameter. Once connected, the daemon replies with its new status, or an error message.
The app must contact the registration API every time the user connects. If the daemon returns an error following the connect request, the app can keep the authentication token for up to 5 minutes. After that time, the app must obtain a new registration token.

The app should check the status of the daemon at least once every 5 seconds.
To check the status of the networking daemon, the GUI app sends a `get_status` request to the daemon, which replies with its current status, or an error message.

To disconnect the networking daemon, the GUI app sends a `disconnect` request to the daemon, which replies with its new status, or an error message.

Cloudflare ZT Client GUI

Expectations and requirements

GUI: We expect, at minimum, an app that displays a GUI that the user can interact with to connect and disconnect. The GUI must accurately reflect the status of the daemon, and updates periodically (at minimum every 5 seconds).

Platform: The app must run on at least **one** major platform, such as:

- macOS 13 and above; and/or
- Windows 10 and above; and/or
- Ubuntu 22.04 LTS

To be clear, you may write an app that only runs on one of these platforms.

Programming language: You can also use the languages and frameworks of your choice. For example, you can use Swift and SwiftUI for macOS only. Or C# and WPF for Windows only. Or C++ and GTK for Linux and macOS. Or Flutter for all 3 platforms. Or Rust and tauri for all 3 platforms too.
We expect you to use the language and framework you're the most familiar with, so that you know the idioms best, and do not have to spend too long learning the platform for this task.

README: You should provide a README file that includes

Paid software: You may not use software or 3rd party components that would require the interview panel to buy them to compile, run, or test your code, unless you can provide a licence for the panel. As an exception, you may use Visual Studio for Windows.

Compilation: Your app must compile and be free of errors and warnings. Make sure you provide instructions on how to build the client in the README.

3rd party components: If the source code requires additional components, you must provide a README to list the dependencies and how they are to be installed.
You are free to use 3rd party components, but be careful of the licence they ship with.
You can use as many or as few 3rd party components, but try to keep it small if possible.

Documentation: Document the code the way you would at work. Hint: we like really well documented code.

Tests: Your code should be as testable as possible. The interview panel must be able to run the unit tests on their machine. All unit tests must pass successfully.

Visuals: We are more interested in your code than in your design skills. You do not have to spend time to make the GUI good-looking (but can if you so wish).

UX: While we don't expect the GUI to look incredible, we do expect a good user experience. The GUI should be easy to use, intuitive, and informative.

Quality: We pay great attention to quality, and expect a bug-free, crash-free application.

Cloudflare ZT Client GUI

Non-requirement bonuses

We do not expect you to do more than what's required. Candidates can absolutely be successful without implementing any of the following suggestions.
We have enough to do in our day-to-day!

But if you fancy adding a bit of ~~cloud~~ flair, why not add any of these suggestions?

- Make the app send notifications (via the OS) on status changes
- Make the app run headless, and only show e.g. as a popover from the menu bar on macOS, the system tray on Windows, status icons on GNOME, etc.
- Package the app so it's ready to be installed (e.g. pkg on macOS, msi on Windows, etc.)

Cloudflare ZT Client GUI

Submission

Once you are done, please share the source code with the recruiter, who will share it with the interview team.

You can share your code through any file storage.
Please ensure to share everything that is required to compile, run, and test your application.
If 3rd party components are required, either provide them, or provide instructions in the README on how to get them.

If you choose to upload your code to online platforms such as GitHub, GitLab, etc. you can help us keep the hiring process fairer by keeping the repository locked down (private).

Cloudflare ZT Client GUI

Specifications

Registration API specifications

GET `https://warp-registration.warpdir2792.workers.dev/`

Retrieves the temporary authentication token

Request

Path parameters

None

Headers

Header	Description
`X-Auth-Key`	The key to authenticate this call with. Use the value 3735928559

Response

Response codes

HTTP Code	Available fields
`200`	`status, data`
`401`	`status, message`
`500`	`status, message`

Response fields

Name	Type and possible values	
`status`	`<string> “error” | “success”`	
`message`	`<string?>` (optional, only on error)	
`data`	`<object RegistrationResponse?>` (optional, only on success)	
	Name	**Type and possible values**
	`auth_token`	`<number>`

Cloudflare ZT Client GUI

Examples

Success

```
C/C++
curl --header 'X-Auth-Key: 3735928559'
https://warp-registration.warpdir2792.workers.dev/
{
  "status": "success",
  "data": {
    "auth_token": 245346444925233
  }
}
```

Authentication failure

This error is returned if you do not pass a valid `X-Auth-Key` in the headers.

```
C/C++
curl https://warp-registration.warpdir2792.workers.dev/
{
  "status": "error",
  "message": "Invalid authentication key"
}
```

Other errors

```
C/C++
curl --header 'X-Auth-Key: 3735928559'
https://warp-registration.warpdir2792.workers.dev/
{
  "status": "error",
  "message": "An active incident is currently affecting this
API (ICDT-1234). Please consult our status page for more
details: https://www.cloudflarestatus.com/"
}
```

Cloudflare ZT Client GUI

Daemon specifications

To communicate with the mock daemon, you use UNIX sockets on macOS and Linux, or named pipes on Windows.

When the daemon starts, it creates a socket/pipe on which it listens. The daemon only ever writes to the socket to reply to requests. It never writes to the socket on its own.

For ease of testing, the daemon takes a few parameters. You can find them by running the help command.

```C/C++
$ ./daemon-lite --help
Usage: daemon-lite [OPTIONS]

Options:
  -f, --failure-rate <FAILURE_RATE>
          How often the requests should fail (once every X calls). Use 1 to
always fail. Use 0 to always succeed [default: 4]
  -c, --connect-timeout <CONNECT_TIMEOUT>
          The maximum amount of time the connect request can take to
establish a connection (in milliseconds) [default: 5000]
  -d, --disconnect-timeout <DISCONNECT_TIMEOUT>
          The maximum amount of time the disconnect request can take to
disconnect (in milliseconds) [default: 3000]
  -h, --help
          Print help
  -V, --version
          Print version
```

The provided daemon doesn't actually connect to any server or make any network connection. It only serves as a simulation.

While the binary is called daemon, you don't need to run it as a daemon/service on your machine. You can simply run it in a terminal and leave it to run while you develop.

Cloudflare ZT Client GUI

Communicating with the daemon

The daemon listens on this socket path:

```c
/tmp/daemon-lite
```

All requests and responses consist of two parts:

1. The first 8 bytes contain the 64-bits integer indicating the size of the payload that follows
2. The following bytes contain the payload, in JSON format

When writing to the socket, you MUST always first send/write 8 bytes for the size before writing the JSON payload.

When reading from the socket, you MUST always read 8 bytes to know how many bytes to read for the JSON payload.

The endianness for the size is that of the platform the daemon is running on (on macOS, little endian).

See the python script below for an example on how to communicate with the daemon.

Cloudflare ZT Client GUI

Example python script

Python
```python
import os
import json
import socket
import struct

SOCKET_PATH = "/tmp/daemon-lite"

with socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) as sock:
  sock.connect(SOCKET_PATH)

  send_payload = {
    "request": {
      "connect": 245346449271196
    }
  }
  # Serialise the payload into a JSON string
  send_payload_json = json.dumps(send_payload)
  # Determine the size of the JSON payload
  send_payload_size = len(send_payload_json)
  # Convert the size of the JSON payload into an array of 8 bytes
  # (which represents a 64-bits value)
  send_payload_size_bytes = send_payload_size.to_bytes(8, byteorder='little')

  # -- Sending the request --

  print("Sending ", send_payload_json)
  # First, send the size of the JSON payload to the socket
  sock.sendall(send_payload_size_bytes)
  # Then, send the JSON payload to the socket
  sock.sendall(send_payload_json.encode())

  # -- Reading the response --

  # Read 8 bytes (64-bits integer) for the payload size from the socket
  # This tells us how much to read from the socket for the response payload
  recv_payload_size_bytes = sock.recv(8)
  # Convert the array of bytes into an integer. This becomes the size of the
response payload to read
  recv_payload_size = int.from_bytes(recv_payload_size_bytes,
byteorder='little')
  # Read the JSON payload of the response
  recv_payload_json = sock.recv(recv_payload_size)
  # Deserialise the JSON payload into a python object. This is the response!
  recv_payload = json.loads(recv_payload_json)
  print(f"Received response: {recv_payload}")
```

Cloudflare ZT Client GUI

Requests

Name	Type and possible values	
request	<enum DaemonRequest>	
DaemonRequest (enum, only one value is possible)	**Name**	**Type and possible values**
	connect	<number> (Auth token retrieved from API)
	disconnect	
	get_status	

Examples

Connect request

C/C++
```
{
  "request": {
    "connect": 245346437489485
  }
}
```

Disconnect request

C/C++
```
{
  "request": "disconnect"
}
```

Get status request

C/C++
```
{
  "request": "get_status"
}
```

Cloudflare ZT Client GUI

Response

Name	Type and possible values	
status	<string> “error” | “success”	
message	<string?> (optional, only on error)	
data	<object DaemonResponseData?> (optional, only on success)	
Daemon Response Data	**Name**	**Type and possible values**
	daemon_status	<string> “connected” | “disconnected”
	message	<string?> (optional, only if disconnected because of an error)

Examples

Daemon is connected

C/C++
```
{
  "status": "success",
  "data": {
    "daemon_status": "connected"
  }
}
```

Daemon is disconnected

The daemon is disconnected without an error following, for example, the disconnect request.

C/C++
```
{
  "status": "success",
  "data": {
    "daemon_status": "disconnected"
  }
}
```

Cloudflare ZT Client GUI

Daemon is disconnected following an error

The request is successful, because the status was retrieved successfully, but the daemon disconnected because of an error.

C/C++
```
{
  "status": "success",
  "data": {
    "daemon_status": "disconnected",
    "message": "The remote server closed the connection"
  }
}
```

Daemon is unable to fulfil the request

C/C++
```
{
  "status": "error",
  "message": "Unable to retrieve status: daemon is busy"
}
```

Daemon can't connect

Following a connect request, the daemon may time out connecting to the remote server.

C/C++
```
{
  "status": "error",
  "message": "Connection to the remote server timed out."
}
```