OpenOBD

Cody Emrick
A14740612

Christopher Madrigal
A15702734

June 10th, 2020

Contents

1 Abstract

Vehicles seem to perpetually exist in a bygone era. By the time new cars adopt a technology, whether it be radio, CD players, or navigation systems, there's a very good chance that a similar feature has been available elsewhere for years beforehand. And when you want to upgrade, you will need to buy a new car to do so. Worst of all, you'll likely end up with a proprietary, custom solution with limited features. Shopping for a car based on the quality of the infotainment system is impractical, and even high-end cars are subject to this problem. Likewise, home mechanics that want access to diagnostic data will need to purchase a specialized, manufacturer-specific device just to read data. All vehicles since the mid-1990's come equipped with a standardized onboard diagnostic (OBD) system, but manufacturers and professional mechanics have done everything they can to lock consumers out of these vital systems. Our project proposes to break their monopoly on your car's data by allowing users and developers a safe, easy way to access and utilize data to enhance your vehicle's functionality and allow for modular upgrades.

2 Introduction

2.1 The Problem

Fundamentally, the core problem here is that most vehicles do not operate on a standard platform. Even similar models by the same manufacturer might only have a few parts in common, and even when a manufacturer creates a standardized platform, it's more for their benefit than the customer's and eventually they will do a refresh that will break compatibility. This forces aftermarket parts to be built around a specific make and model, perhaps with some flexibility based on the year or platform. Regardless, this has led to a quagmire of compatibility issues. For an aftermarket part manufacturer, there is little incentive to release an upgrade for a car nobody is buying new anymore.

We can further sub-divide this problem into two classes: physical and electron. For example, if you want to buy a new radio for your car, you need to find one that both fits physically into your dashboard and also will connect to your car's audio system. On modern cars, your "infotainment" system may also contain many of the controls for your vehicle, or also display select diagnostic information, which means it must interface with whatever wiring and information specification the manufacturer provides. Our goal will be to tackle the latter issue, and we believe that, free of a requirement to match proprietary data connections, it will be easier to adapt a generic product to fit physically in a variety of vehicles. Even on modern vehicles with an Apple CarPlay or Android Auto are often difficult to upgrade and will not have access to a full range of information, although they are a step in the right direction.

Currently, the market is basically limited to infotainment systems. But we imagine that, with a unified platform to provide access to data, it might be possible to expand the market for other devices which can display data. It could be possible to project navigation and speedometer data directly onto your windshield, so you can keep your eyes on the road and also see where you are going. You could wear augmented reality glasses which highlight your car's location so you never get lost in a parking lot again. Your music player could change which playlist it's using based on your speed or location. You could be notified of an engine misfire when it happens. Your oil pressure could be read directly. You could use a machine learning algorithm to train yourself to drive more economically. We originally pitched ideas just like these, but kept running into the same issue: there is no standardized way to access your car's data for applications such as these, and if you developed a stack just for interfacing with the car then you could have to pick-and-choose which to use.

2.2 The Goal

We believe that someone looking to upgrade their vehicle should simply be able to go online or to a store, browse products that solve a problem they have, and then be able to use this in their car. They should not have to be locked-in to a given vendor or system, nor should they have to fear that they will lose access to their data or control of their car. As vehicles trend increasingly towards being "smart", "Internet of Things" objects, the issues inherent in the modern smart-device ecosystem creep in to a traditionally very stable platform, and nobody should have to worry about their car being disabled or damaged remotely via exposure to a broader network.

Additionally, as creators, we want more freedom to get data from our cars and to experiment with solutions to the problems with have. This is the basis of innovation, and currently any individual hoping to create a new product will have to navigate a technical minefield just to develop a one-off product for their own vehicle. A single man working alone ought to be able to create a solution and share this with others in whatever way they see fit. This is a fundamental aspect of “hacker culture”, and by lowering the bar to creating novel solutions we hope to spur innovation in this market.

Additionally, we think it’s important that any third-party devices not have to compete over usage of the OBD-II port. By providing this as a modular part of a larger theoretical stack, free other devices to worry about doing their particular job correctly.

2.3 Our Solution

However, all vehicles since the mid-1990’s come equipped with the OBD-II specification for onboard diagnostics. This federally-mandated system provides a common interface for storing and accessing data relevant to your car’s performance. Some manufacturers choose to build on top of this system to expand functionality.

With the above problems and goals in mind, we created the following guidelines for our device:

It Must Be Open: In order for users to build on our work and audit its security, the platform must be open. A closed platform just adds one more inflexible, untrustworthy option.

It Must Be Secure: We want to authenticate communications between the user and their vehicle. While secrecy of the data is important, ensuring that requests come from and responses are returned to the owner of the vehicle and their authorized devices only is absolutely the most important thing. If the platform is insecure, it could leak private information about the user to malicious third-parties, or in a worst-case scenario, cause damage to their vehicle.

It Must Be Safe: The platform itself should reliably interface with vehicles without causing any harm. A developer should not be concerned that a bad write will brick their car’s diagnostic computer. They must trust it to intercept and sanitize inputs to prevent mistaken or malicious commands.

It Must Be Expandable: The core set of OBD-II functions is standard across all vehicles, but we must allow for users to support additional functionality their manufacturer may provide. Additionally, users who want to write their own software or make their own hardware should be able to easily prototype on top of our platform.

With these goals in mind, we were able to take a common-sense approach to the design of our device, dubbed OpenOBD, which would allow us to interface easily with other devices while providing reliable service.

The Unix Philosophy is as follows:

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface.

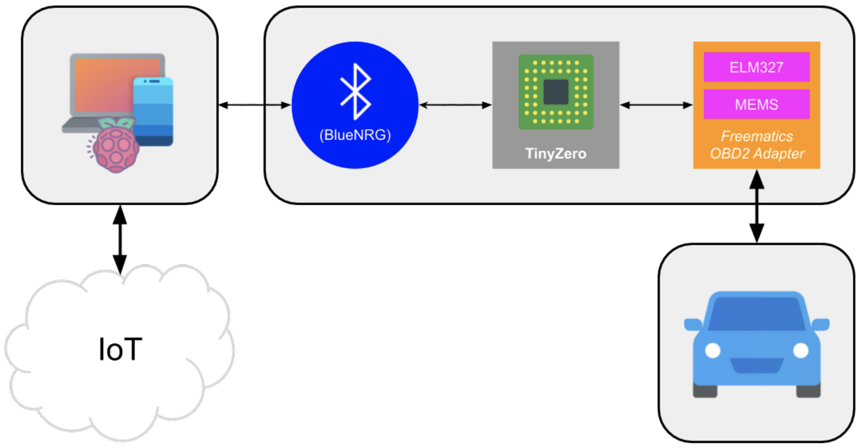
We wish to expand this philosophy into the domain of the Internet of Things.

3 Technical Material

3.1 Prior Work

Plenty of devices exist on the market today for connecting smartphones to OBD systems. Consequentially, there are apps that support dozens of OBD adapters and that share diagnostic information with the user. However, not all adapters require secure pin authentication, and seldom do any supporting apps perform any extra authentication. Further, while the OBD technology employed by any of these solutions is industry standard, the modules and apps themselves do not adhere to any particular standard. Not every app developer can target every module, and are often times limited by what the host platform supports. For instance, iOS devices are far more limited in terms of what the user may do with Bluetooth, and often the practical solution for developers is to use an ad-hoc wifi network instead.

3.2 Design



The system is a relay between the car's OBD port and any supporting Bluetooth host. In this case, a supporting host is one that parses UART text streams. In particular, the text stream is comprised of valid JSON objects. We chose JSON because it is internet-ready, seeing as how RESTful APIs largely employ JSON in their communication schemes. Our decision is not without precedent, as many smart devices communicate using JSON (for example, Philip's Hue wireless lights).

For communicating with other devices, we chose Bluetooth. It's the most common option for device-to-device connections, and modern Bluetooth with Low Energy (BLE) is quite efficient, which is important for a device which can potentially leech your car's battery. Ad-hoc Wi-Fi is just not efficient nor particularly secure, and interfacing over it would have been unnecessarily difficult for no additional advantages. It was important that consumer smartphones be able to interface directly with the OpenOBD module.

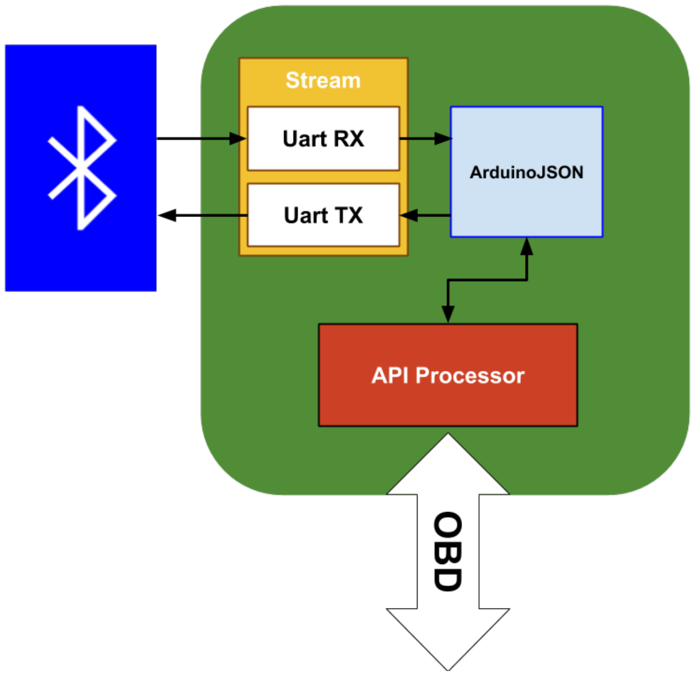


Figure 1: The datapath in closer detail

3.3 Platform

For our Microcontroller, we selected the TinyZero by TinyCircuits. This is an ARM microcontroller that is modular and designed for prototyping. It is very small, which makes it ideal for prototyping a project that needs to fit into a compact package. The platform has excellent support for many embedded libraries, especially those designed for Arduino, and it supports a modular system of “stacking” modules, whereby a number of separate add-ons can simply be attached to each other, like a simplified Arduino HAT system. This makes it perfect for expanding functionality as we continue to prototype.

One decision we had to make was whether we implement our own OBD-II communicator. We ultimately chose not to do this for a variety of reasons. The OBD-II specification covers how data should be stored and transmitted, as well as what kind of data should be tracked and made available. A number of different, competing specifications for actually retrieving this data exist, and they vary greatly in voltage, pinout, and data format. This makes interpreting the data difficult and potentially hazardous to the vehicle if not done correctly. It would also have been more work than the rest of the project combined, and likely take up the majority of the time. Since commercial solutions for this connector exist, we decided it was better to focus on the novel aspects of our project and not to re-invent the wheel. After a survey of the market, we found two decent options, both of which use an ELM347 chip to handle the bulk of the work. After comparing options, a dongle by Freematics was cheaper and provided additional functionality built right in, so it became the obvious choice. Unfortunately, it did not arrive before the end of the quarter, likely due to shipping delays.

4 Milestones

Read Data: Physically read data from a device.

Write Data: Physically write data to the device.

Expose Data: Expose the data via an API

Example Application: Create a simple app that showcases how to use the API. Can also be used for debugging purposes.

Encrypt Data: Take this datapath and add user authentication.

Allow Modifying Vehicle State: Expose safe writes to the vehicle.

Additional Sensors: Add additional sensors to the device which cannot be found on many cars, such as GPS, accelerometer, etc.

4.1 Read Data

While we did not have a chance to test this (see the issues section below), we were able to use the ELM driver provided by Freematics for use with their dongle to implement wrappers which should read and write the data. The ELM utilizes UART to communicate with the microcontroller. A series of PIDs can be transmitted, and then a read will be performed via the OBD-II port. The result is then returned and passed to another wrapper to format a JSON response packet.

4.2 Write Data

Similar to Read Data, we could not test this, but we have implemented a dummy function in accordance with the documentation.

4.3 Expose Data

Our system is built around an API which processes and returns JSON structures. We expose the data by providing a clean, consistent specification for what fields to send and what sort of response to expect. Instead of having to know every PID that is implemented, you simply send a request for the fields you would like read and they, or an error, will be returned. This works, and while it is missing some of the expanded functionality of the Freematics dongle, the standard PIDs are all supported.

4.4 Encrypt Data

Upon activating the device, you will need to pair your device with it. This is done by generating an RSA keypair. The "public" key is then sent over (encrypted) Bluetooth to your device. This certificate can be shared to other devices you want to have access to your OpenOBD dongle. This provides authentication of the user/device. The Bluetooth spec itself provides encryption based around a PIN which the user must enter when it establishes a connection. This way, the data is encrypted and all identities can be verified. Since the device will automatically decrypt and encrypt with its private key, data sent elsewhere will be useless and requests from unauthorized users will likewise be invalid commands.

4.5 Example App

We have a functional Android application which can send requests and receive a response. It currently just prints out the returned JSON. It is merely a proof-of-concept for sending and receiving formatted JSON, and while it is a minimum viable product (MVP), it serves its purpose for debugging and providing sample code to would-be programmers who wish to build on our platform.

4.6 Allow Modifying Vehicle State

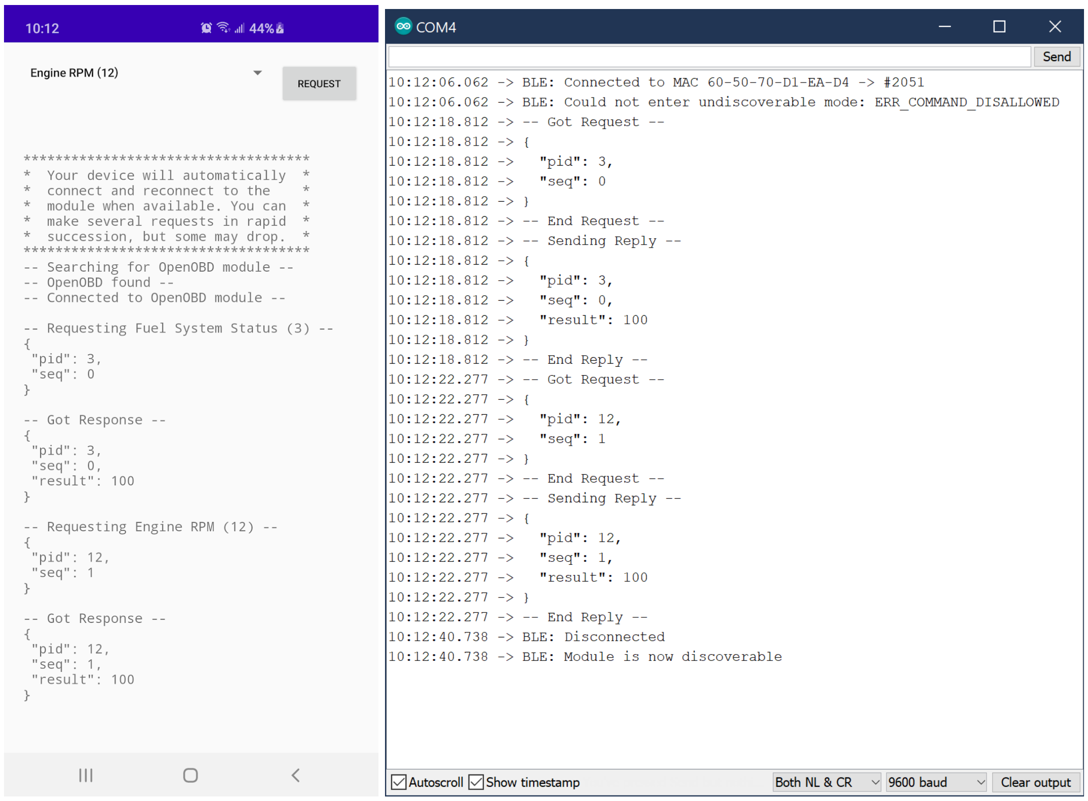
This is mostly unimplemented. While we have a specification for how to format a JSON write query, there is currently no wrapper to conduct writes on the OpenOBD platform. The reason for this is that it is relatively low-priority. There are only a few writes you often want to do, such as clearing an error code, but writing is also the biggest source of vulnerabilities. Implementing a few simple writes would be possible, but it was decided to focus on other pathways and expand this later. Sanity-checking inputs is important.

4.7 Additional Sensors

We have added a GPS module and a storage module to our TinyZero. This allows tracking the vehicle's location remotely. We have plans to add a SIM Card module, to allow vehicle status to be tracked via the internet, but we do not have one of these modules. The Freematic UART dongle also allows for sensing battery levels, and we have implemented preliminary support for this, though it has gone untested. This and a few other optional PIDs exist as part of the ELM chip.

5 Results

Our platform, though incomplete, serves as a functional proof-of-concept for our vision. While it cannot quite be described as an MVP, this is solely because we are missing the connector and have no been able to test it on a real vehicle. The entire datapath to read and transmit data securely has been implemented. We have dummied the UART response from the ELM and have tested the secure connection over Bluetooth.



```
*************************************
*  Your device will automatically  *
*  connect and reconnect to the    *
*  module when available. You can  *
*  make several requests in rapid  *
*  succession, but some may drop.  *
*************************************
-- Searching for OpenOBD module --
-- OpenOBD found --
-- Connected to OpenOBD module --

-- Requesting Fuel System Status (3) --
{
 "pid": 3,
 "seq": 0
}

-- Got Response --
{
 "pid": 3,
 "seq": 0,
 "result": 100
}

-- Requesting Engine RPM (12) --
{
 "pid": 12,
 "seq": 1
}

-- Got Response --
{
 "pid": 12,
 "seq": 1,
 "result": 100
}
```

```
10:12:06.062 -> BLE: Connected to MAC 60-50-70-D1-EA-D4 -> #2051
10:12:06.062 -> BLE: Could not enter undiscoverable mode: ERR_COMMAND_DISALLOWED
10:12:18.812 -> -- Got Request --
10:12:18.812 -> {
10:12:18.812 ->   "pid": 3,
10:12:18.812 ->   "seq": 0
10:12:18.812 -> }
10:12:18.812 -> -- End Request --
10:12:18.812 -> -- Sending Reply --
10:12:18.812 -> {
10:12:18.812 ->   "pid": 3,
10:12:18.812 ->   "seq": 0,
10:12:18.812 ->   "result": 100
10:12:18.812 -> }
10:12:18.812 -> -- End Reply --
10:12:22.277 -> -- Got Request --
10:12:22.277 -> {
10:12:22.277 ->   "pid": 12,
10:12:22.277 ->   "seq": 1
10:12:22.277 -> }
10:12:22.277 -> -- End Request --
10:12:22.277 -> -- Sending Reply --
10:12:22.277 -> {
10:12:22.277 ->   "pid": 12,
10:12:22.277 ->   "seq": 1,
10:12:22.277 ->   "result": 100
10:12:22.277 -> }
10:12:22.277 -> -- End Reply --
10:12:40.738 -> BLE: Disconnected
10:12:40.738 -> BLE: Module is now discoverable
```

Autoscroll: ✓ Show timestamp: ✓ Both NL & CR 9600 baud

Figure 2: The example Android application and the corresponding serial output from the prototype module.

Preliminary results for testing security were good. If one assumes a malicious agent is able to steal the user's Bluetooth PIN or otherwise exploit a flaw which allows them to sniff the Bluetooth packets, they still cannot decode a message encrypted with either of the RSA keys. Likewise, a phony command without the proper authentication is ignored. However, security is non-trivial and ideally, if we were to launch this as a real product, we would want to have our implementation audited professionally.

Though we are using dummy data instead of the actual UART connection to read data from a real vehicle, the device can process requests rapidly. We expect the ELM to be the bottleneck for requests, however, and we are eager to see what kind of throughput we can get on a real vehicle.

The device's size, even in prototype form, is small enough to fit within an OBD-II port. The connector itself is significantly larger, and we believe with a dedicated Printed Circuit Board (PCB) we could make it even smaller. This would make it the smallest such device on the market.

6 Future Work

First and foremost, we still need to attach and test the Freematics OBD Dongle. This will complete our desired deliverables and produce an MVP. Beyond this, We would really like to add SIM Card support so that the module can be tested for internet readiness (i.e. testing if the module can reliably send JSON data to a designated internet host). This would enable the module to be accessed remotely, and we deliberately chose an authentication scheme that would make this possible. Further, we aim to support the expanded diagnostics capabilities offered by some manufacturers (e.g. BMW and Mercedes) despite being proprietary. Supporting these will be much more difficult, as they may change between models, so we may also need to implement a way users can develop custom PIDs and either flash their modules or customized payloads to enable utilization of these diagnostic systems. Finally, to encourage further support of the API, we wish to distribute libraries to simplify interfacing with the API and to expose support in our demonstration app for Tasker and IFTTT, two popular automation apps for Android and iOS respectively.

7 Conclusion

In all, the module is a neat and well-defined proof on concept, but we regretfully have no tests for a real-world setting. We still have a lot to learn about bringing a product to market. The module offers plenty of room for further improvement, which remains one of our key goals as we continue to introduce new iterations of our design. We think ours could be a viable product, and it would certainly find use by its developers.

In terms of meeting our goals, I feel that the design of the product is a success. By the metric imposed above, we created a platform which is (theoretically) open, secure, and extensible. It is open by virtue of being open source, and currently runs on a hardware platform anyone can buy and use. If we brought this to market, we would like the hardware to be open source, too. It is secure because all interactions are authenticated by generated keypairs, and the device will wipe any remaining data in the event it is reset. It is currently safe, although this is because potentially unsafe operations have not been implemented, but because requests are done via an intermediary API, we can ensure all calls made are safe. And finally, it is extensible, because it just provides a data pathway. Developing an app or designing hardware to interact with your car can be no harder than one that accesses social media or scrapes a website.

At the very least, both of us developed a product that solves a problem we had. Our solution is no-compromises, beats out the alternatives in a number of areas, and we were able to learn a lot by building it. Although we had a number of unfortunate setbacks this quarter, we do feel we met our benchmarks for success and accomplished what we set out to do.