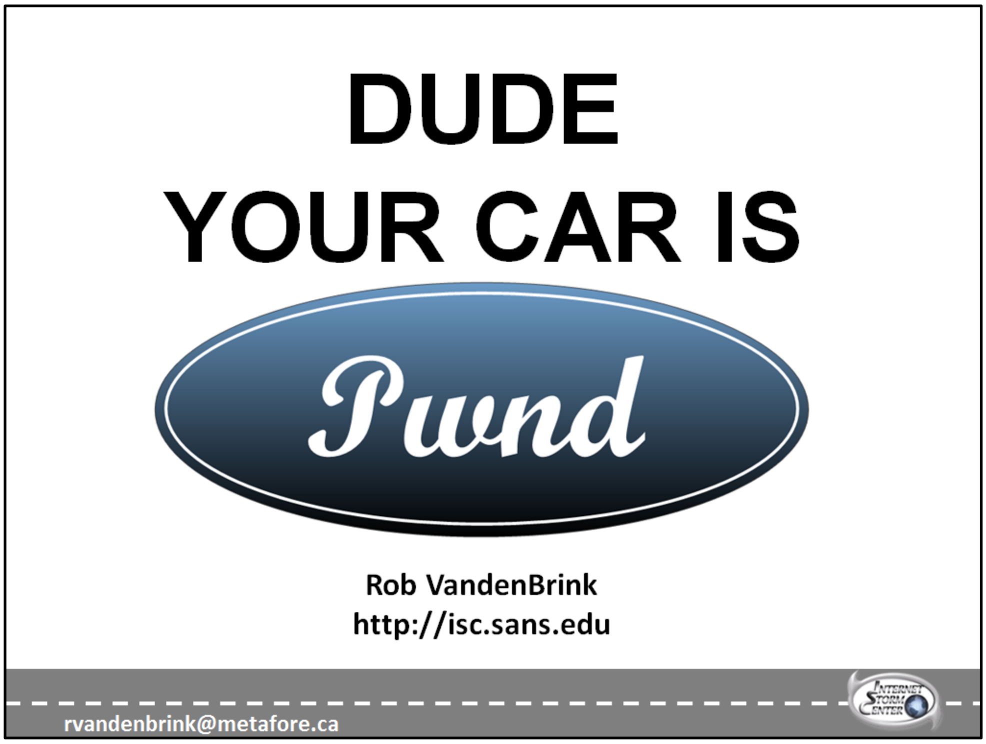
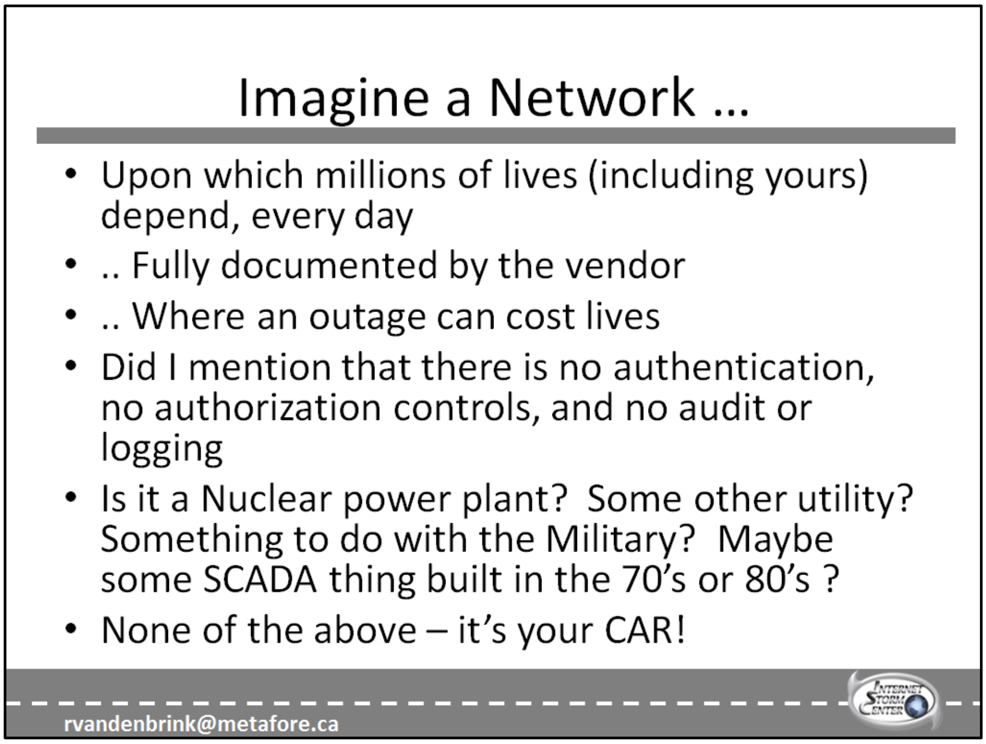


DUDE YOUR CAR IS Pwnd

Rob VandenBrink
http://isc.sans.edu

rvandenbrink@metafore.ca


Imagine a Network …

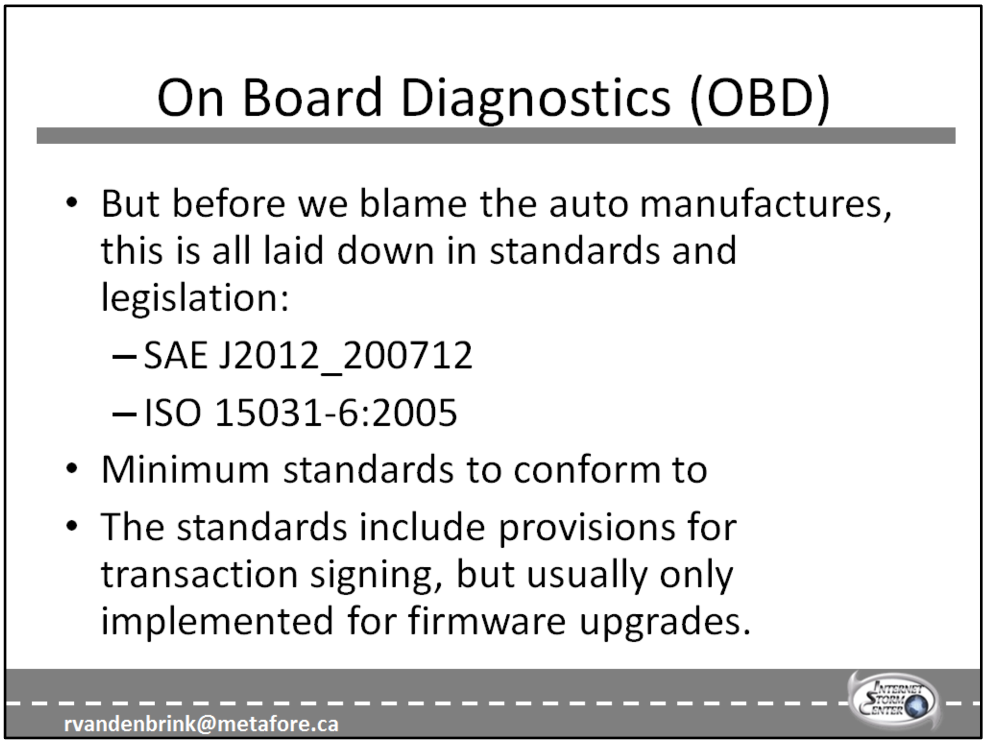
- Upon which millions of lives (including yours) depend, every day
- .. Fully documented by the vendor
- .. Where an outage can cost lives
- Did I mention that there is no authentication, no authorization controls, and no audit or logging
- Is it a Nuclear power plant? Some other utility? Something to do with the Military? Maybe some SCADA thing built in the 70's or 80's ?
- None of the above – it's your CAR!

rvandenbrink@metafore.ca

We concentrate our efforts on "critical infrastructure" on the big, obvious utility networks – power and water plants and networks, as well as on the internet backbones and the like. I think we're missing the boat a bit – there are numerous "networks of little things" that also, in aggregate, add up to large critical infrastructure networks – the network in our cars being one of those. In fact, on it's own, the network inside my car is as critical (or more so, if I happen to be driving at the time) as power to my house.

Unfortunately, the On Board Diagnostic (OBD) network in our cars is completely open , completely documented, and is being pushed more and more to open, documented and unauthenticated wireless access.

We'll discuss this, current security work and future trends in this presentation, with some nifty (if I do say so myself) python code and demos !



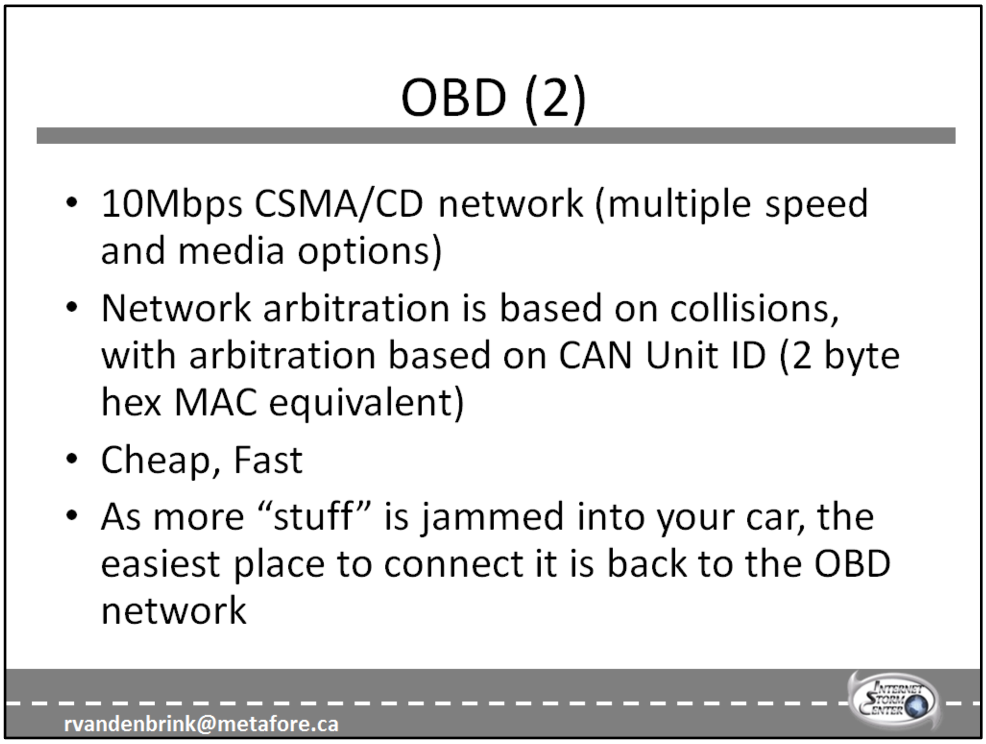
On Board Diagnostics (OBD)

- But before we blame the auto manufactures, this is all laid down in standards and legislation:
  - SAE J2012_200712
  - ISO 15031-6:2005
- Minimum standards to conform to
- The standards include provisions for transaction signing, but usually only implemented for firmware upgrades.

rvandenbrink@metafore.ca

The OBD (On Board Diagnostic) network in our individual cars follow the Henry Ford tradition of mass production. Everyone's OBD network must conform to a standard protocol, with a minimum set of supported instructions. This minimum set is defined in SAE Standard J2012-200712, and is mirrored exactly in the ISO Standard 15031-6:2005 (latest version).

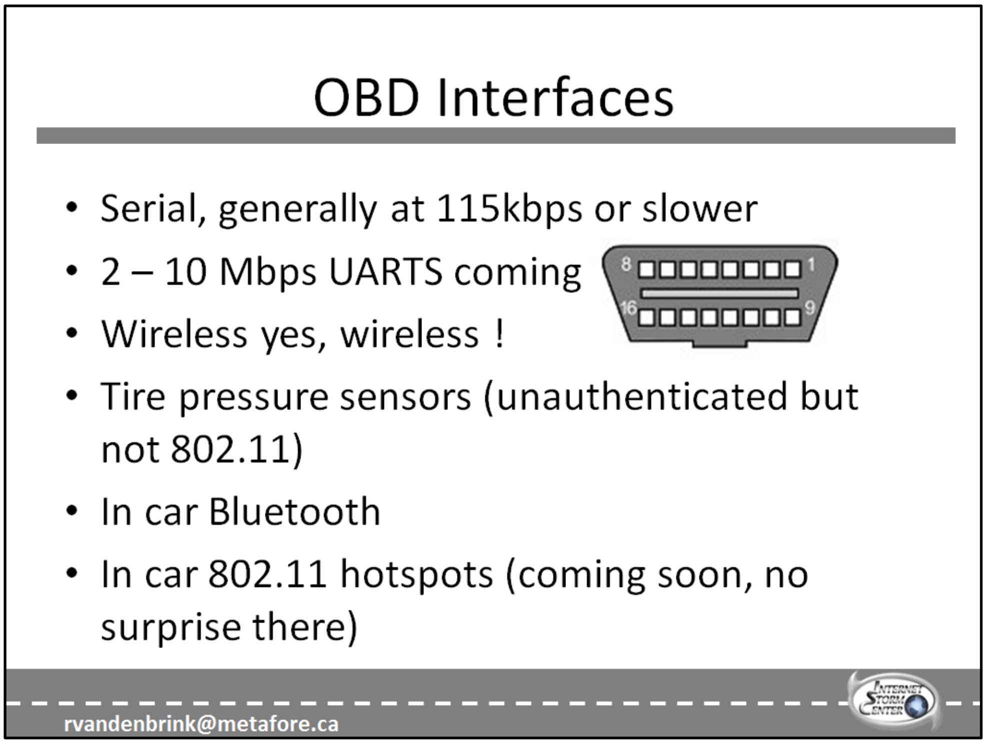
Light duty vehicles (less than 8,500 lbs) in North America have been legislated to implement OBD-II since 1996, Medium Duty Vehicles (8,500 – 14,000 lbs) since 2005 and Heavy Duty Vehicles (heavier than 14,000 lbs) since 2010.


OBD (2)

- 10Mbps CSMA/CD network (multiple speed and media options)
- Network arbitration is based on collisions, with arbitration based on CAN Unit ID (2 byte hex MAC equivalent)
- Cheap, Fast
- As more “stuff” is jammed into your car, the easiest place to connect it is back to the OBD network

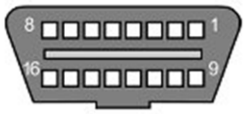
rvandenbrink@metafore.ca

So, what does the OBD network look like? Like a slower, dumber ethernet (sorta).
It’s CSMA / CD (Collision Sense Multiple Avoidance / Collision Detect) – with arbitration based on sender Unit or CAN ID (more or less the MAC address). For instance, Ford uses a CAN ID of 720 for the dashboard instrument cluster
It’s cheap, it’s fast, and manufacturers seem compelled to hang everything off the network. It seems to be human nature that if there’s a network at hand, everything close to it MUST be connected to it.



OBD Interfaces

- Serial, generally at 115kbps or slower
- 2 – 10 Mbps UARTS coming
- Wireless yes, wireless !
- Tire pressure sensors (unauthenticated but not 802.11)
- In car Bluetooth
- In car 802.11 hotspots (coming soon, no surprise there)



rvandenbrink@metafore.ca

The traditional interface to the OBD network for enthusiasts and researchers is via a big, ugly serial interface – the connector is roughly 4cm across, so it's not like any serial interface you're familiar with. It's usually located under the dash, often directly under the steering wheel. This interface can, depending on the SAE network standard being used, be as slow as 4800bps, but is usually interfaced to at 115kbps. Speeds up to 2mbps can be attained with the proper interface hardware.

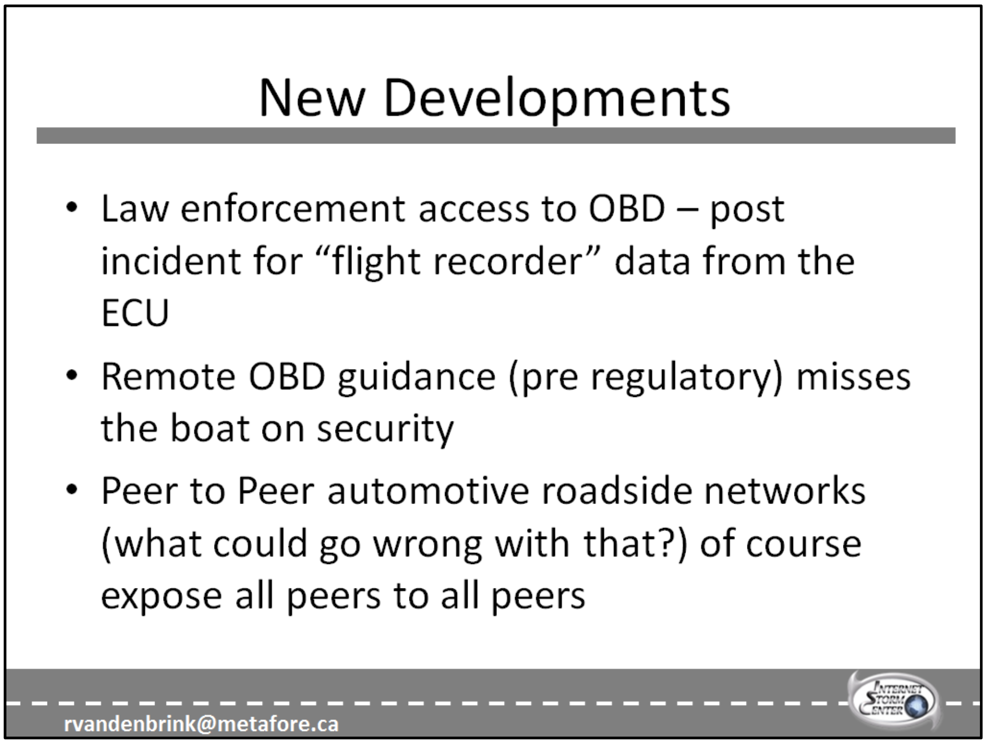
The most common interfaces to this serial port are based on the ELM327 chipset, and run anywhere from $20 to $200, depending on speed, features and buffering.

Unfortunately, the nominal speed of the network is 10Mpbs, and buffering is minimal, so when larger datasets are being processed this needs to be accounted for - more on this later.

Recent updates to the standards force all modern cars to have wireless Tire Pressure Monitor Sensors (TPMS). This standard has been on the books since 2001, and legislated since 2005 (

Fortunately, the TPMS link is not an 802.11 wireless link.   It's a proprietary wireless protocol, usually at 315MHz. Unfortunately, you can buy a sensor reader for around $300. These readers can be easily torn down and repurposed as a general purpose OBD interface, capable of wireless reads and writes to the in-car network. So, security by obscurity, nothing new to us here.

What’s worse is the trend we’re seeing more and more today. Cars with Bluetooth, cars with 802.11 hotspots might seem innocuous – what’s so bad about interfacing your phone to your car you say? Remember what we said about folks networking everything they can lay their hands on? If you think that your car’s bluetooth only talks to the dash and speakers, you’re sadly mistaken !


New Developments

- Law enforcement access to OBD – post incident for "flight recorder" data from the ECU
- Remote OBD guidance (pre regulatory) misses the boat on security
- Peer to Peer automotive roadside networks (what could go wrong with that?) of course expose all peers to all peers

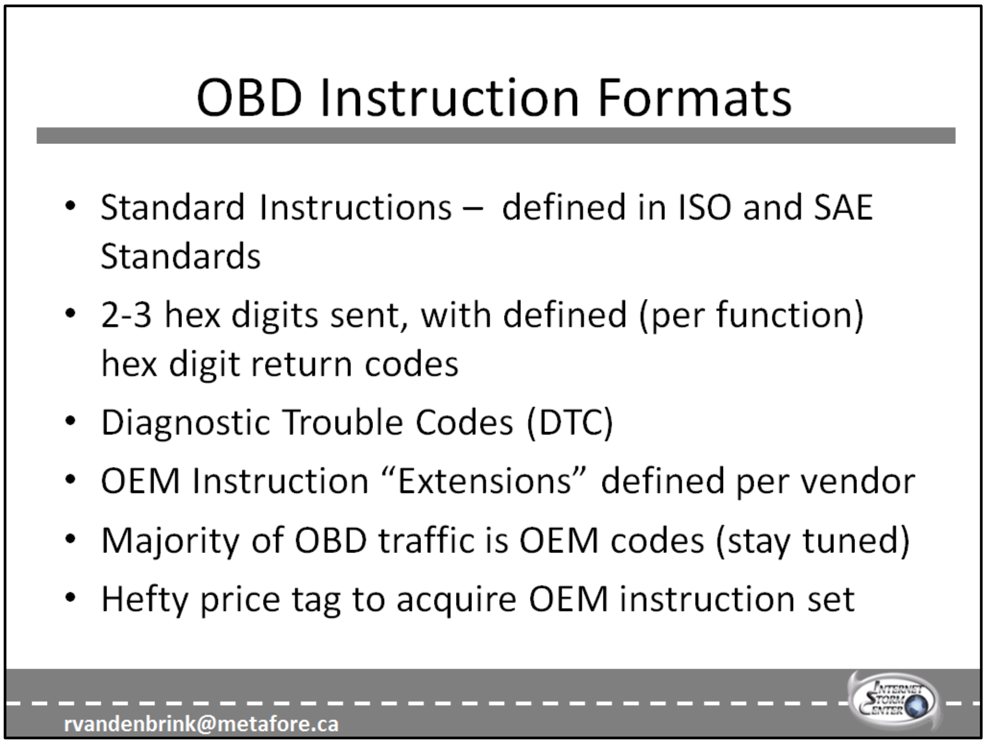
rvandenbrink@metafore.ca

So we've painted the picture it's pretty bleak for security on your car network. What's coming.
You'd think things would be getting better, but no, it's getting worse.

A current concern is that while Law Enforcement has not used the "flight recorder" functions on the ECU (Engine Control Units), there is nothing stopping them from doing that. On the good side, after an accident they could record speed, accelerator and brake activity leading up to the event. On the bad side, they could cruise the parking lot at the mall, or residential streets, and look for folks who were speeding prior to being parked.

The current guidance on Remote OBD (ie – roadside collection of OBD data from moving cars) does have a chapter on Security. However, it's entirely concerned with people falsifying their OBD readings, for instance to make their emmissions appear compliant when in fact they are not (by clearing codes for instance). There is a brief section on putting a password on any database of collected information, but not even a hint on authentication or encryption of the wireless communications.

Even worse, there is loads of research on Peer-to-Peer roadside networks. As in peer to peer between moving cars, including fixed position roadside wireless stations. The thinking is that these will be used for analyzing traffic patterns, automated braking for accident avoidance, and even heads-up display advertising for local businesses. Again, this research is all function based, focusing on how cool data collection and accident avoidance would be, and how the range of these networks could be extended, operated at lower power, and made more reliable with better handoffs as network members move at high speed relative to each other. Again, if you think there is a ton of research on security challenges in a mobile network, think again.



OBD Instruction Formats

- Standard Instructions – defined in ISO and SAE Standards
- 2-3 hex digits sent, with defined (per function) hex digit return codes
- Diagnostic Trouble Codes (DTC)
- OEM Instruction "Extensions" defined per vendor
- Majority of OBD traffic is OEM codes (stay tuned)
- Hefty price tag to acquire OEM instruction set

rvandenbrink@metafore.ca

What does an OBD Instruction look like?
It's usually a 2 or 3 Hex digit code, with a 6-10 byte response code.

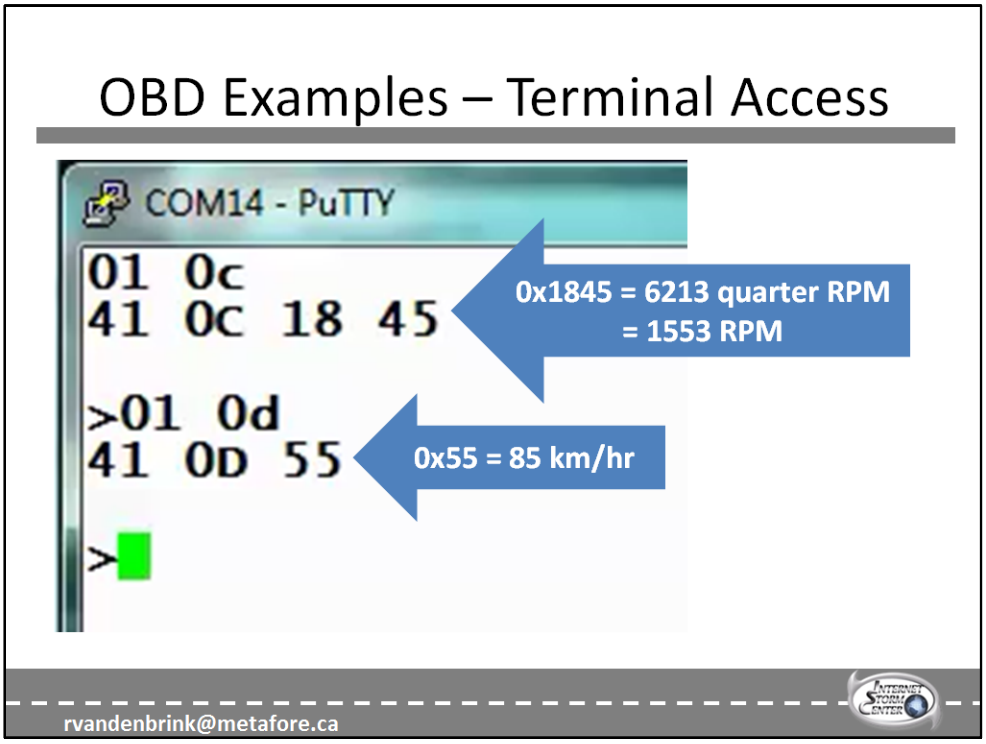
Diagnostic codes are usually much simpler, where you are querying for a static "trouble code" rather than a dynamic status of a running component.

SAE / ISO "Standard" OBD codes all start with "01"
Each manufacturer has their own set of OEM Extensions to OBD. These extensions all start with different digits (for instance, Ford uses 21 and 22 a lot). A typical vehicle will implement 30-40 ISO codes, and 200-300 OEM codes.

OEM extensions are typically not public. The manufacturers charge real dollars for access to their codes.

For instance to get codes for GM, Honda, Suzuki or BMW:
GM $50,000
Honda $5,000
Suzuki $1,000
BMW $7,000 plus $1,000 per update. Updates occur every quarter

Ford publishes their codes via the ETI (Equipment and Tool Institute) – http://www.etools.org. ETI has dues of $5,000 per year.



OBD Examples – Terminal Access

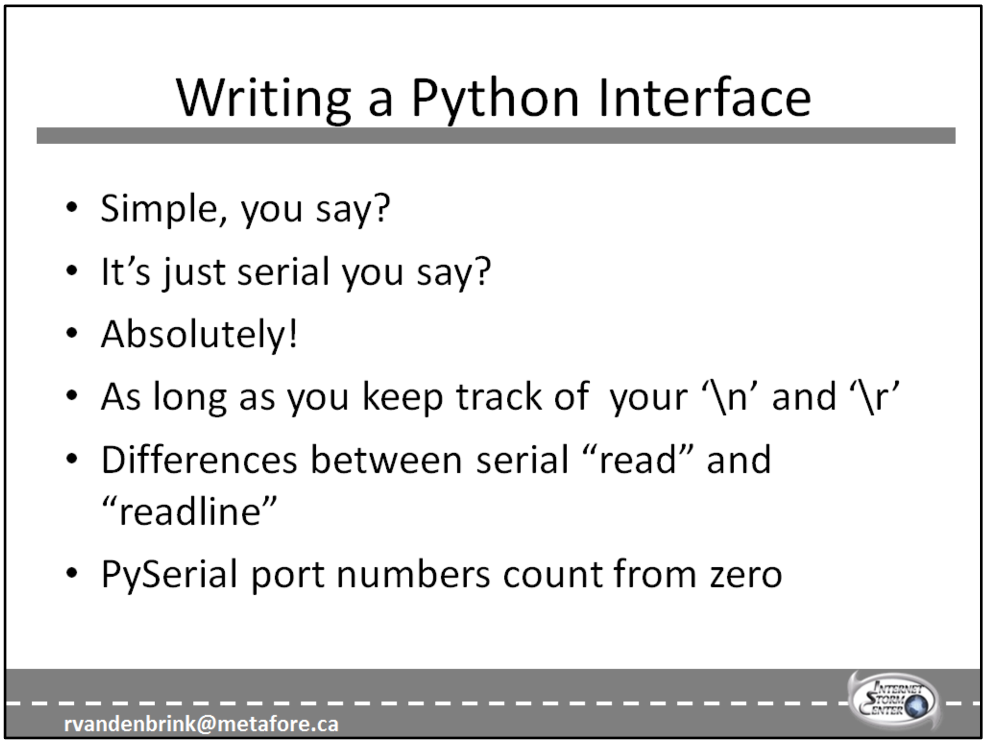
```
01 0c
41 0C 18 45

>01 0d
41 0D 55

>
```

In this example, we'll use putty on the serial port (COM14) and query for the RPM and Speed.
The RPM value is returned in quarter revolutions per minute, so must be divided by 4 for the final result.

As discussed, all sent and received values are in ASCII hex representation, so usually need to be converted back to decimal.



Writing a Python Interface

- Simple, you say?
- It’s just serial you say?
- Absolutely!
- As long as you keep track of your ‘\n’ and ‘\r’
- Differences between serial “read” and “readline”
- PySerial port numbers count from zero

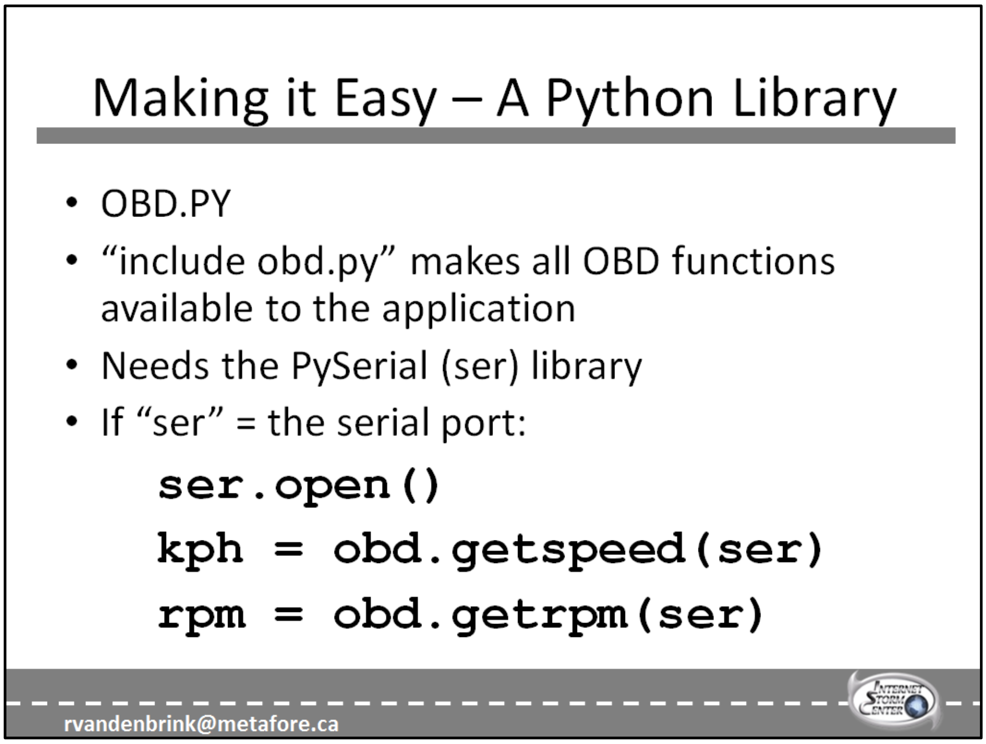
rvandenbrink@metafore.ca

So this seems like a really simple interface to program against. And it actually is – sort of.

The main thing to keep track of is the Idiosyncrasies of the language and libraries you are writing in. Differences between \r and \n are especially important.

In Pyserial for instance, “readline” wants to see a \n before it will return a value.
In a time-sensitive application like an OBD based dashboard, it’s much safer to execute a read(80), to read the buffer, to a maximum of ‘xx’ characters, then process that. If no characters are there, or if there are 10 or 20 characters, at least the read completes immediately and passes control back to the next instruction, rather than waiting forever for a ‘\n’ that might never come

Pyserial also (most correctly and properly) counts serial ports from 0. So COM14 is actually port 13 for Pyserial.



Making it Easy – A Python Library

- OBD.PY
- "include obd.py" makes all OBD functions available to the application
- Needs the PySerial (ser) library
- If "ser" = the serial port:

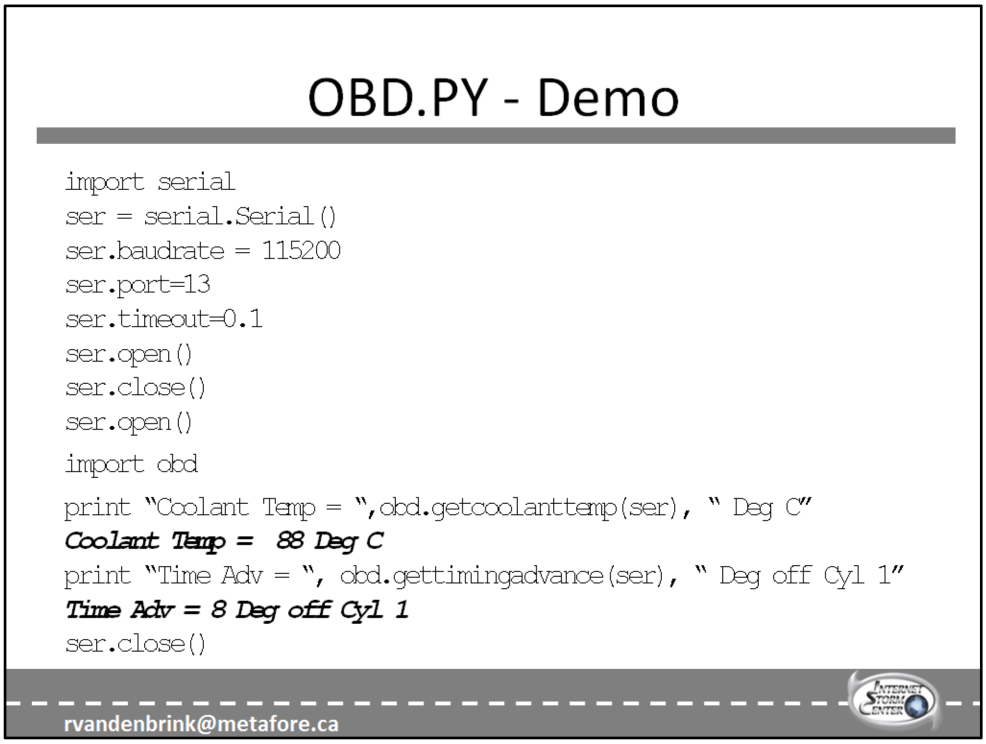
```
ser.open()
kph = obd.getspeed(ser)
rpm = obd.getrpm(ser)
```

rvandenbrink@metafore.ca

As you might have guessed, we're getting to "the code part"

Setting all the OBD queries up in a Python library made the final applications much cleaner. Each OBD call is in the library, **obd.py**

All the serial calls are done by pyserial. The example snip here shows two calls to get the speed and RPM.



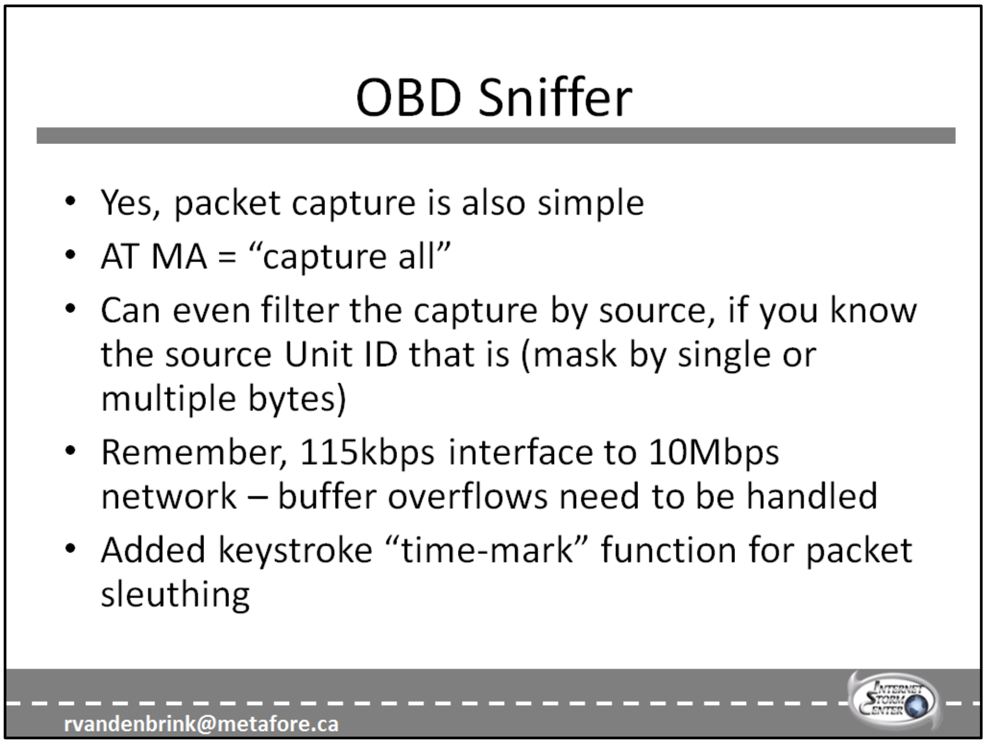
OBD.PY - Demo

```
import serial
ser = serial.Serial()
ser.baudrate = 115200
ser.port=13
ser.timeout=0.1
ser.open()
ser.close()
ser.open()
import obd
print "Coolant Temp = ",obd.getcoolanttemp(ser), " Deg C"
Coolant Temp =  88 Deg C
print "Time Adv = ", obd.gettimingadvance(ser), " Deg off Cyl 1"
Time Adv = 8 Deg off Cyl 1
ser.close()
```

rvandenbrink@metafore.ca

This shows a more complete example, which will actually run a short bit of python code using the OBD library

The printed program output is shown in italics



OBD Sniffer

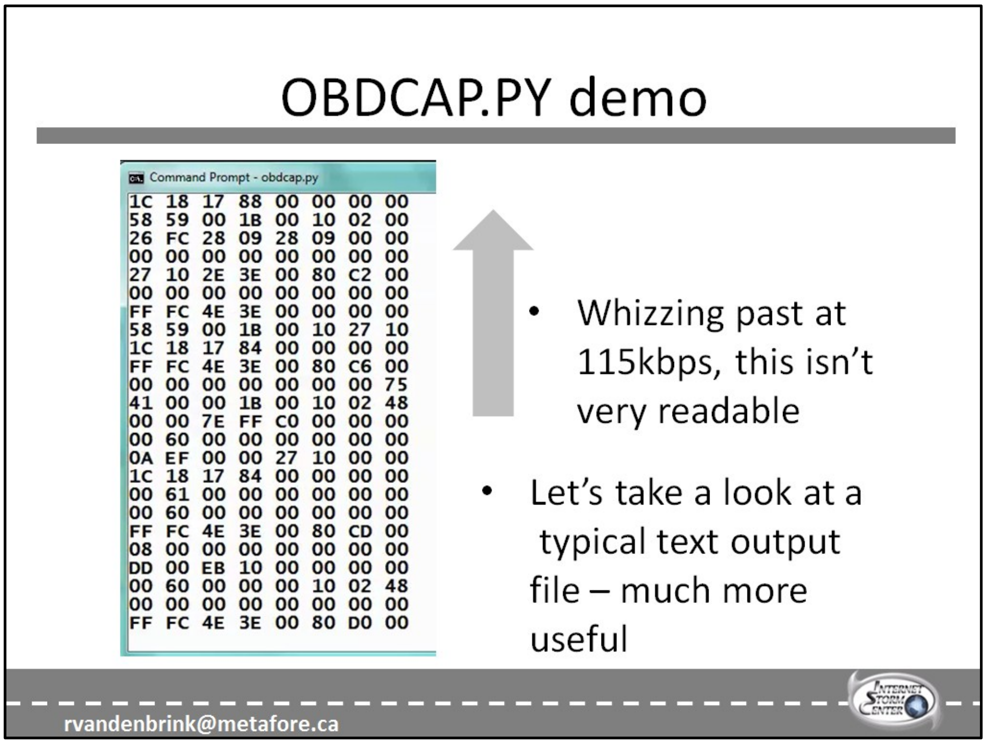
- Yes, packet capture is also simple
- AT MA = “capture all”
- Can even filter the capture by source, if you know the source Unit ID that is (mask by single or multiple bytes)
- Remember, 115kbps interface to 10Mbps network – buffer overflows need to be handled
- Added keystroke “time-mark” function for packet sleuthing

rvandenbrink@metafore.ca

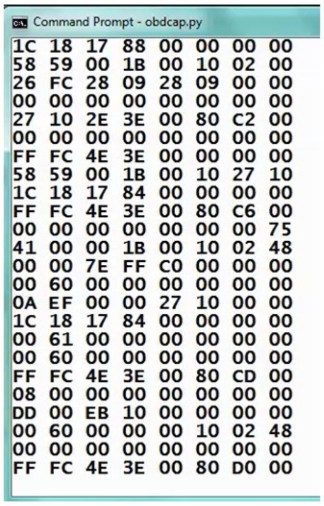
So we can issue commands and get responses – how do we find out about all those other OEM codes we don’t know? What else can we do on this network.

Two birds with one stone – we can write a “packet sniffer” – a short bit of code to capture all the codes traveling on the bus !

I added a “Time Mark” function – when you press a key, a string of Asterixes is added to the capture. This makes it easy to bound an area of traffic with a known function in it (like the door lock command for instance).



OBDCAP.PY demo

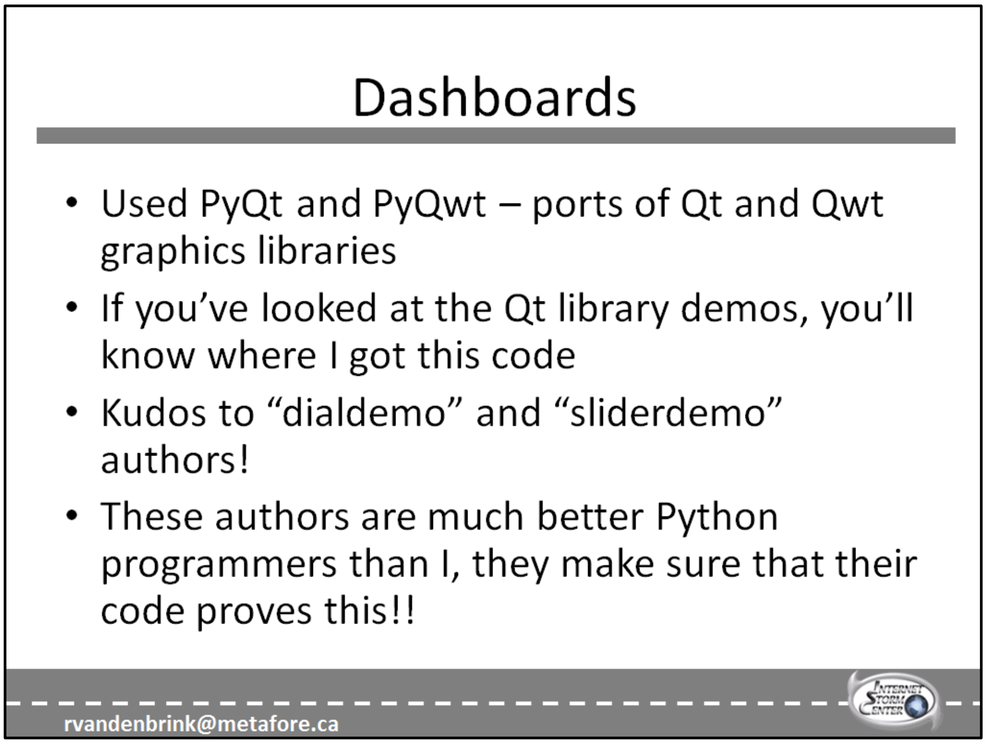


```
1C 18 17 88 00 00 00 00
58 59 00 1B 00 10 02 00
26 FC 28 09 28 09 00 00
00 00 00 00 00 00 00 00
27 10 2E 3E 00 80 C2 00
00 00 00 00 00 00 00 00
FF FC 4E 3E 00 00 00 00
58 59 00 1B 00 10 27 10
1C 18 17 84 00 00 00 00
FF FC 4E 3E 00 80 C6 00
00 00 00 00 00 00 00 75
41 00 00 1B 00 10 02 48
00 00 7E FF C0 00 00 00
00 60 00 00 00 00 00 00
0A EF 00 00 27 10 00 00
1C 18 17 84 00 00 00 00
00 61 00 00 00 00 00 00
00 60 00 00 00 00 00 00
FF FC 4E 3E 00 80 CD 00
08 00 00 00 00 00 00 00
DD 00 EB 10 00 00 00 00
00 60 00 00 00 10 02 48
00 00 00 00 00 00 00 00
FF FC 4E 3E 00 80 D0 00
```

- Whizzing past at 115kbps, this isn’t very readable
- Let’s take a look at a typical text output file – much more useful

rvandenbrink@metafore.ca

As you can see by how fast this whizzes past, it’s much more effective to use I/O redirection to a file, then process the file using grep and the time-mark feature in OBDCAP!



Dashboards

- Used PyQt and PyQwt – ports of Qt and Qwt graphics libraries
- If you've looked at the Qt library demos, you'll know where I got this code
- Kudos to "dialdemo" and "sliderdemo" authors!
- These authors are much better Python programmers than I, they make sure that their code proves this!!

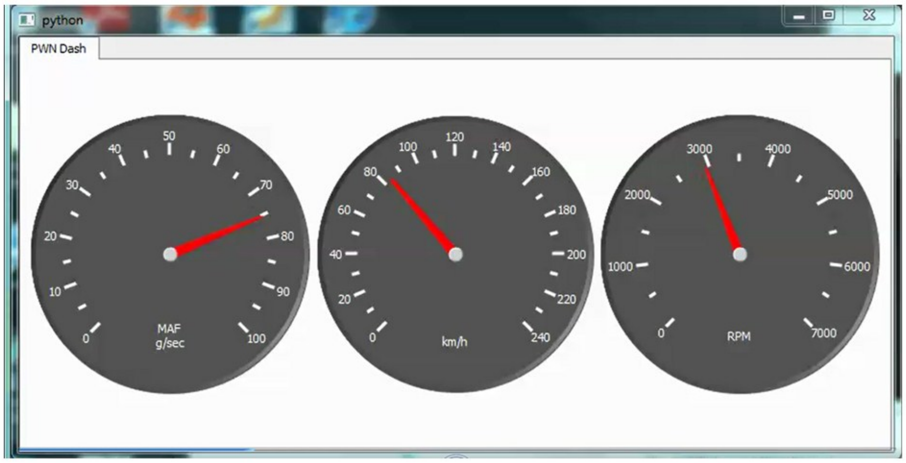
rvandenbrink@metafore.ca

What else can we do?   I wrote a couple of custom dashboards.

The PyQt and PyQwt libraries to do the graphics. They do an admirable job, but you need to be a graduate from the Daystrom Institute to understand the examples included with the libraries!
So you'll note that my code looks suspiciously like the PqQt examples (exactly like in some sections!)

Use was made of "signals" in these examples. Each gauge is tied to a variable. Rather than loop and redraw the gauge, the gauge monitors the variable and redraws itself when a change is detected !   Very neat !

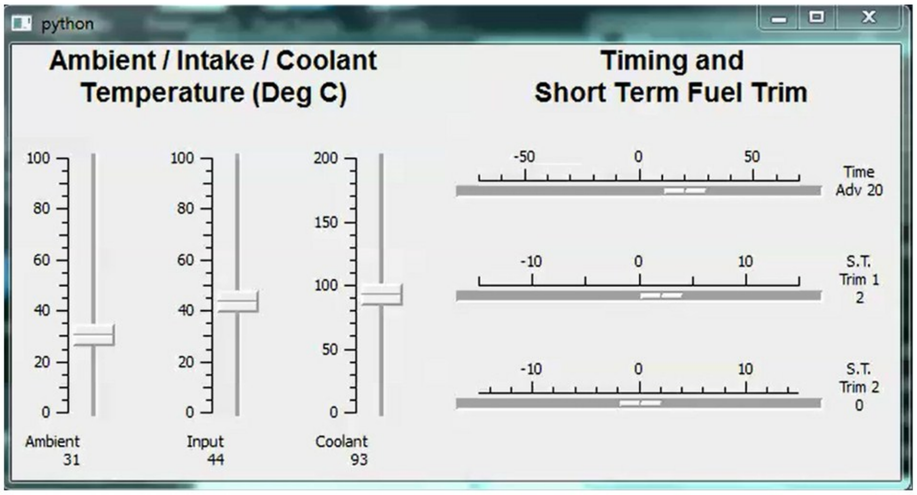
Dashboard – DASH1.PY



rvandenbrink@metafore.ca

DASH1 monitors the Speed in km/hr, the engine RPMs, and the MAF (Mass Air Flow), a measure of the airflow through the injectors in grams/sec

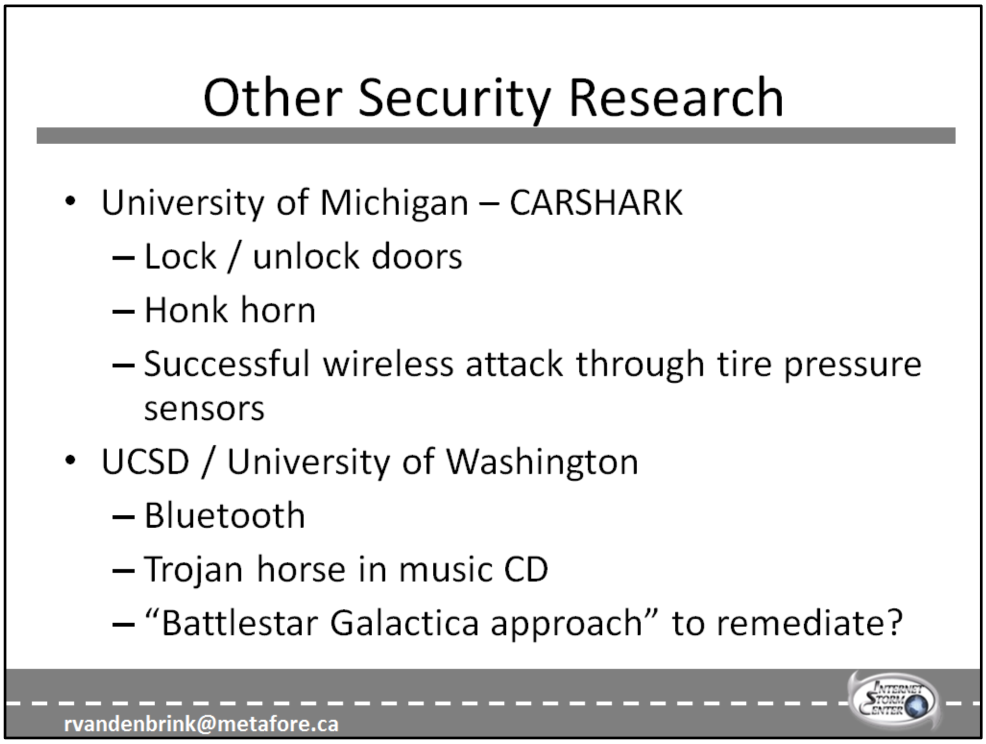
Dashboard – DASH2.PY


python
Ambient / Intake / Coolant Temperature (Deg C)
Timing and Short Term Fuel Trim
Ambient 31
Input 44
Coolant 93
Time Adv 20
S.T. Trim 1 2
S.T. Trim 2 0

rvandenbrink@metafore.ca

Dash2 monitors less “lively” parameters:
The Ambient Temperature in Degrees C
The Intake Temperature in Degrees C
The Coolant Temperature in Degrees C
Spark Timing Advance from Cylinder 1

Short term Fuel Trim – This is the temporary “adjustment” in the closed feedback loop that controls the fuel flow. In my car, there are two Short Term Fuel Trim sensors (both are shown here). There is also a pair of Long Term Fuel Trim Sensors, but unless you’re doing something foolish, you can expect them to usually read Zero. The range for STFT is -100 (Lean) and +100 (rich). The “Trim” value is actually a measurement of the Oxygen Sensor Voltage, and the value is expressed as a Percentage



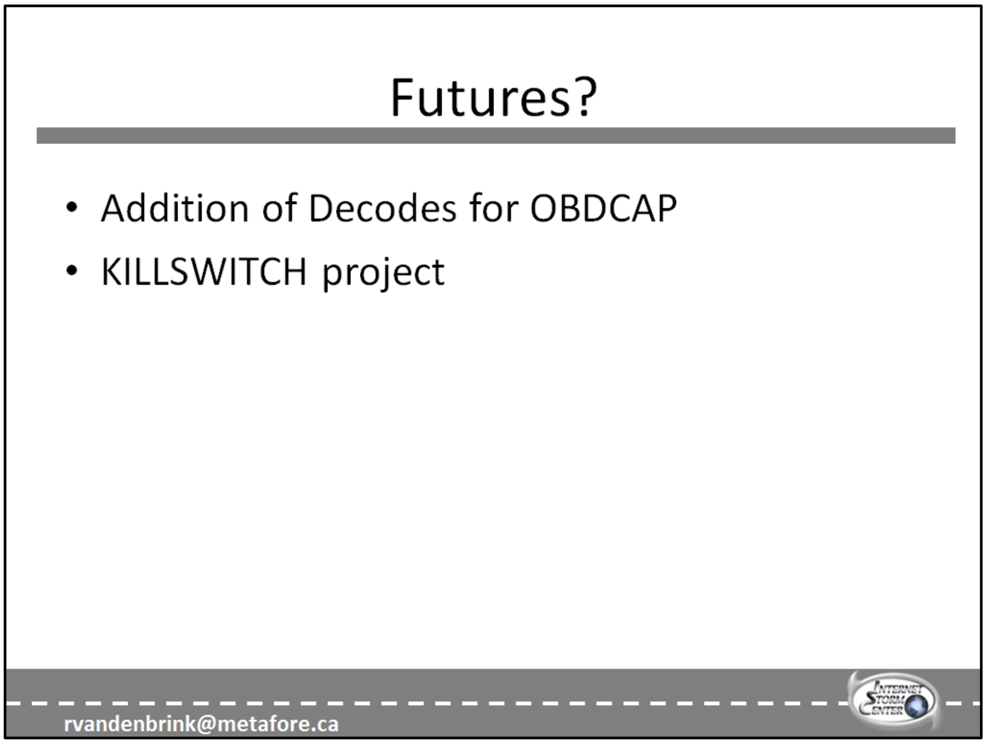
Other Security Research

- University of Michigan – CARSHARK
  - Lock / unlock doors
  - Honk horn
  - Successful wireless attack through tire pressure sensors
- UCSD / University of Washington
  - Bluetooth
  - Trojan horse in music CD
  - "Battlestar Galactica approach" to remediate?

rvandenbrink@metafore.ca

University of Michigan Work:
http://www.autosec.org/publications.html

UCSD / University of Washington
http://it.slashdot.org/story/11/03/12/0114219/Hacking-a-Car-With-Music?from=rss&utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+Slashdot%2Fslashdot+%28Slashdot%29
http://fortcollinswebworks.com/news/car-malware.html
http://www.itworld.com/security/139794/with-hacking-music-can-take-control-your-car



Futures?

- Addition of Decodes for OBDCAP
- KILLSWITCH project

rvandenbrink@metafore.ca

Obviously, you could spend the rest of your days reverse-engineering the OEM codes, then rewriting Wireshark for OBD.
Something tells me I won't be going down this road !

Though writing an OBD plugin for Wireshark might be fun!

But …

Combining some of the concepts from this presentation, think about a tennis-ball sized device, powered by batteries and magnetic. Inside, imagine a small computer – maybe a PWNY, maybe an android device. Combine that with decodes for the ignition and access via the wireless Tire Pressure Sensors, and it should be very possible to make a "killswitch" – a toss-able device that will kill the ignition of any car it sticks to, then lock the doors.

I could see law enforcement applications, but some evil applications also (the police already can do this with OnStar)