Lab1

Attacking an unsecured J1939 network

Remember

There is no 100% security

Security, like all engineering, involves tradeoffs

Know what you are trying to secure

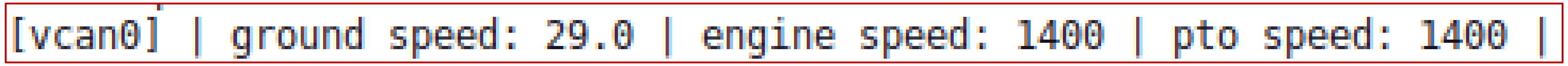
The adversary…

State Sponsored

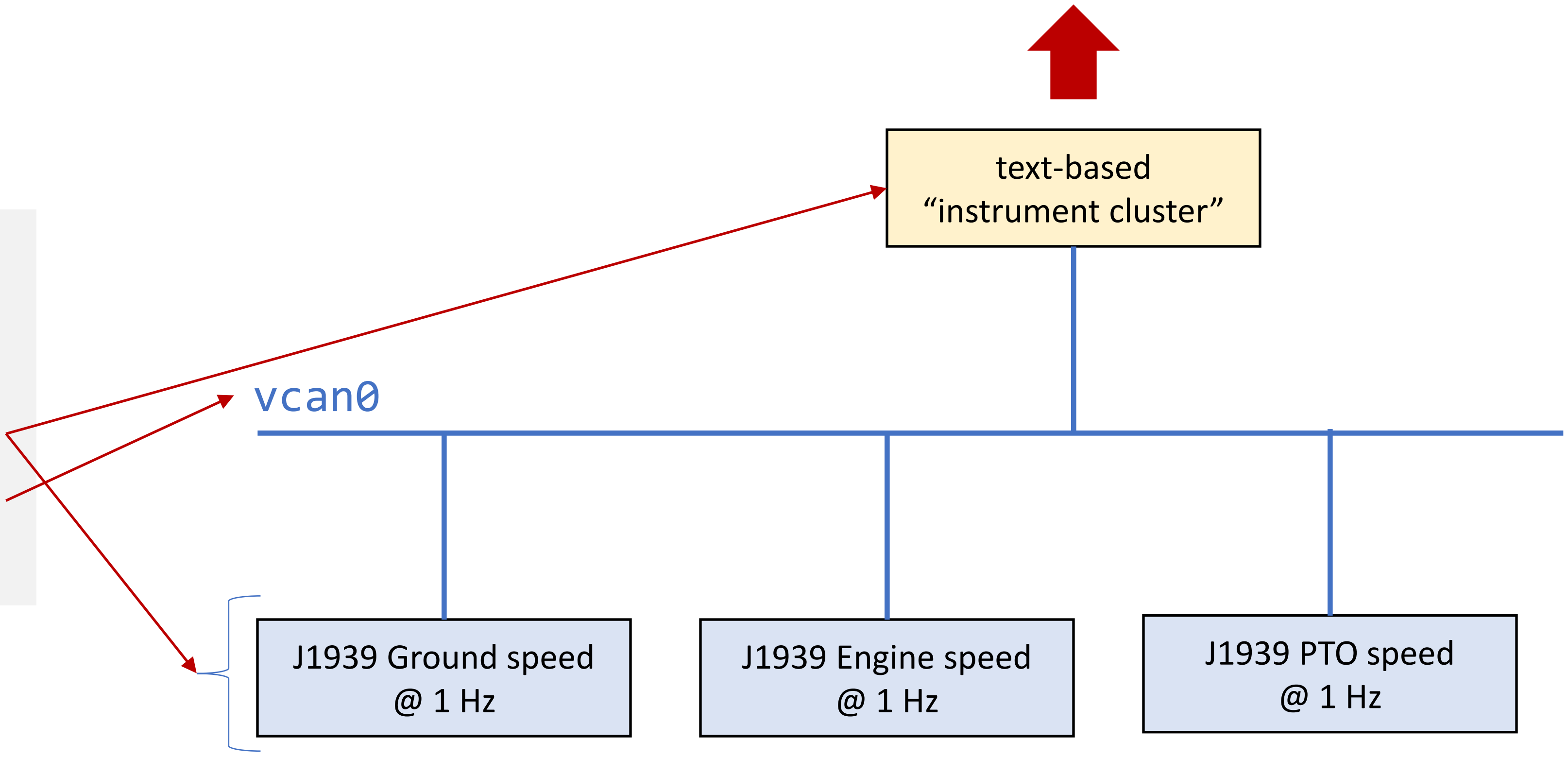
Network Configuration

```
$ tree lab1/bench

.
├── engine_speed.py
├── ground_speed.py
├── launch.sh           (2) Launches ECUs
├── pto_speed.py
├── setup_fd_vcan.sh    (1) creates vcan0
└── text_ic.py
```



```
[vcan0] | ground speed: 29.0 | engine speed: 1400 | pto speed: 1400 |
```



These data sources ramp up and down.

Historical Reference

- Charlie and Chris 2016 – “Jeep 2”
    - Tricked vehicle into doing cyber-physical actions that should have been available only at low ground speed
    - Result: vehicle went into the ditch

https://illmatics.com/Remote%20Car%20Hacking.pdf

https://illmatics.com/can%20message%20injection.pdf

https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/

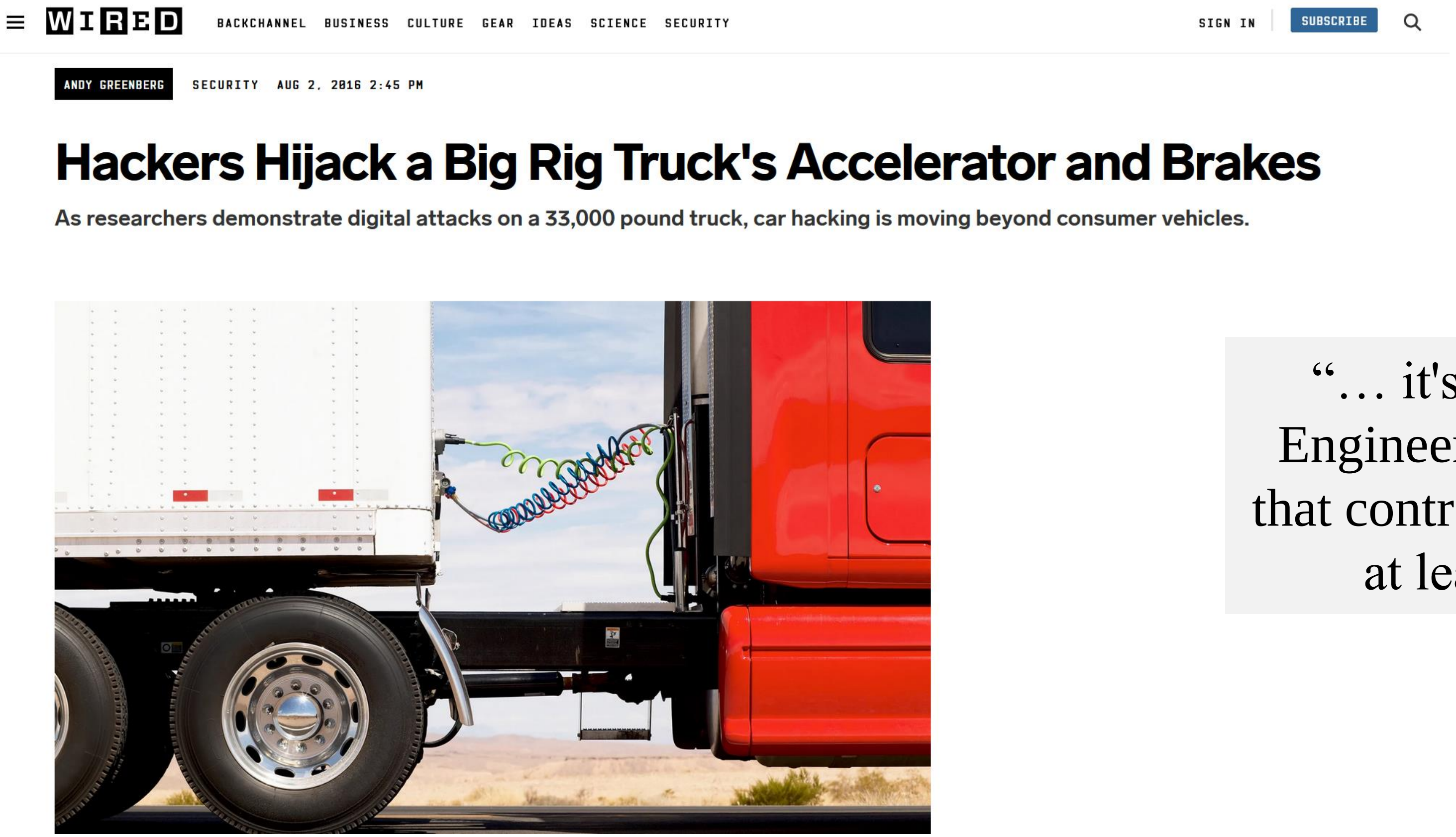
How the New Attacks Work

Instead of focusing on that initial wireless foothold, this time Miller and Valasek wanted to to bypass a set of safeguards deeper in vehicles' networks. Vehicle CAN network components are designed to resist certain dangerous digital signals: The diagnostic mode that Miller and Valasek used to disable the Jeep's brakes, for instance, wouldn't work at any speed above five miles per hour, and the automatic parking assist feature they used to turn its steering wheel only worked when the vehicle was in reverse and traveling at the same low speeds.

But Miller and Valasek have now found techniques to bypass some of those safeguards, with disturbing results. Here's how their new attacks worked: Instead of merely compromising one of the so-called electronic control units or ECUs on a target car's CAN network and using it to spoof messages to the car's steering or brakes, they also attacked the ECU that sends legitimate commands to those components, which would otherwise contradict their malicious commands and prevent their attack. By putting that second ECU into "bootrom" mode---the first step in updating the ECU's firmware that a mechanic might use to fix a bug---they were able to paralyze that innocent ECU and send malicious commands to the target component without interference. "You have one computer in the car telling it to do one thing and we're telling it to do something else," says Miller. "Essentially our solution is to knock the other computer offline."

Historical Reference

- Univ of Mich grad student homework
  - Demonstrated total lack of security in J1939



ANDY GREENBERG SECURITY AUG 2, 2016 2:45 PM

Hackers Hijack a Big Rig Truck's Accelerator and Brakes

As researchers demonstrate digital attacks on a 33,000 pound truck, car hacking is moving beyond consumer vehicles.

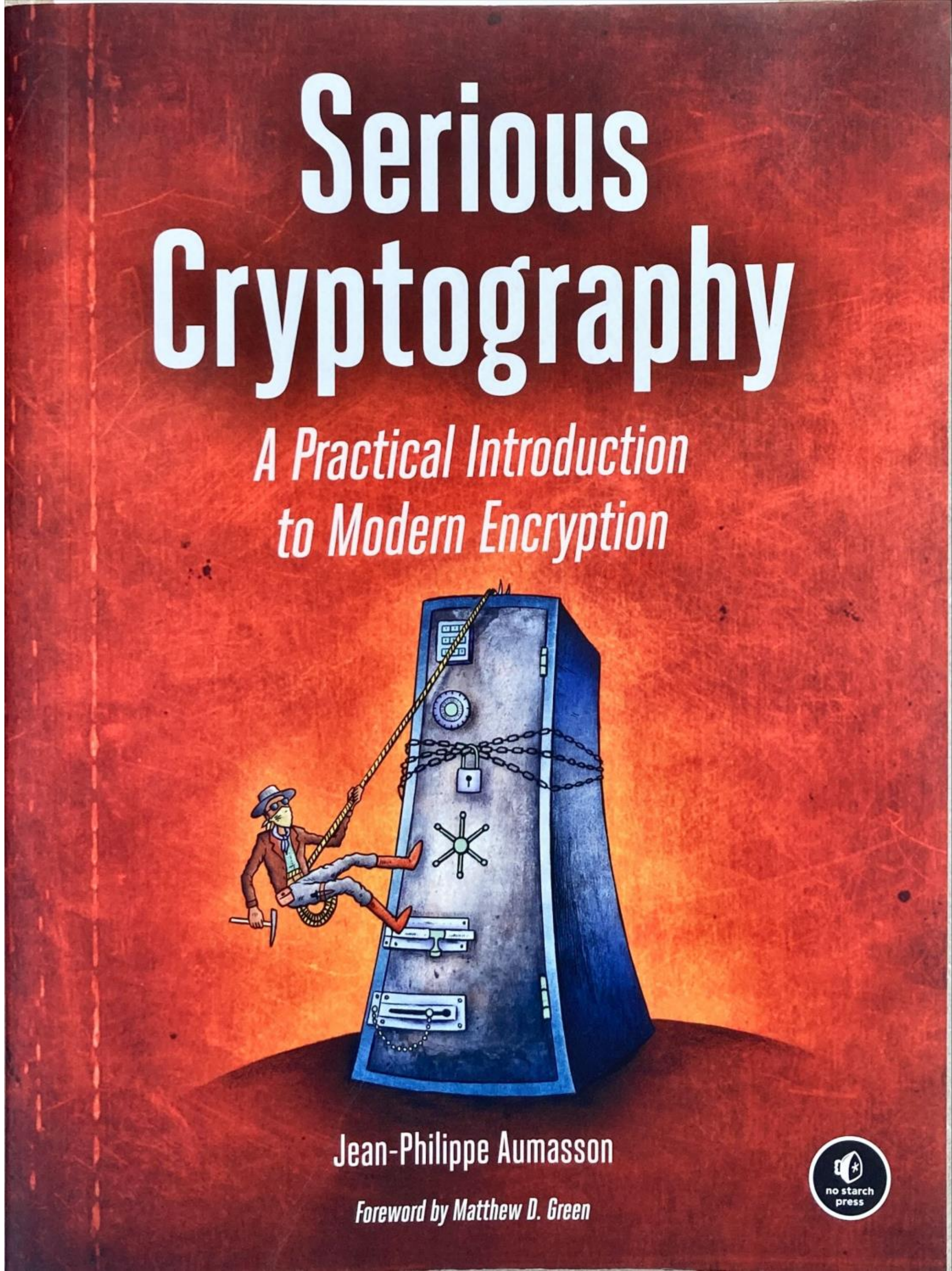
> "… it's the Society of Automotive Engineers (**SAE**), the standards body that controls the **J1939 standard**, that's at least partly responsible…"

article video

Cryptography for the Semester

- AES-128
  - CMAC – message digest
  - CTR (if you get to encryption)
- Curve25519
  - X25519 – key agreement
  - Ed25519 – signing
- NaCl
  - box() and boxopen()
- JWT for credentials

- SHA2
  - SHA256
  - SHA512
  - SHA512 /256


Serious Cryptography
A Practical Introduction to Modern Encryption
Jean-Philippe Aumasson
Foreword by Matthew D. Green
no starch press

Lab – /classroom

```
$ ./setup_fd_vcan.sh
```

Setup the CAN FD network: vcan0

```
> cat -n setup_fd_vcan.sh
     1  sudo modprobe vcan
     2  sudo ip link add dev vcan0 type vcan
     3  sudo ip link set vcan0 mtu 72
     4  sudo ip link set up vcan0
```

Line 1 adds vcan module to linux kernel
Line 2 creates a device ‘vcan0’
Line 3 sets it to be CAN FD w/ mtu
Line 4 starts device ‘vcan0’

Lab – /classroom

```
$ > ./engine_speed.py
```

Starts infinite loop sending engine speeds (ramp-up/ramp-down)

```
> ./text_ic.py
reading a few special J1939 messages from vcan0

use 'cansniffer' on vcan0 to watch all traffic
..........
^c to quit
[vcan0] | ground speed:  0.0 | engine speed:  800 | pto speed:    0 |
```

“text instrument cluster”… reads J1939 messages and converts to engineering units

Lab – /classroom/engine_speed.py

```
> cat -n engine_speed.py

    76  def main():
    77    bus = can.Bus(channel=channel, interface=bustype)
    78    canid = arbid(PRIO, PGN, sa)
    79    speed = 0      # rpm
    80    direction = 1
    81
    82    while True:
    83      # our speed is ramping up and down, forever, 0..4800 rpm
    84      speed = speed + direction * 25
    85      if speed > 4800:
    86        direction = -1
    87        speed = 4800
    88      if speed < 0:
    89        direction = 1
    90        speed = 0
    91
    92      byte4, byte5 = engine2j1939(speed)
    93      data = [0xFF, 0xFF, 0xFF, byte4, byte5, 0xFF, 0xFF, 0xFF]
    94      msg = can.Message(arbitration_id=canid, data=data, is_extended_id=True)
    95      bus.send(msg)
    96      time.sleep(1)
```

Line 82-96 infinite loop ramping up/down the engine speed value and sending message on the bus

Line 92-95 mechanics of building and sending a J1939 frame

Lab – /classroom/engine_speed.py

```
> cat -n engine_speed.py

    28  # return arbitration id
    29  def arbid(priority, pgn, sa):
    30    field1 = (priority << 2) << 24
    31    field2 = pgn << 8
    32    field3 = sa
    33    return field1 | field2 | field3
    34
    35  # convert engine to j1939 2 bytes
    36  # usage: byte4, bytes5 = engine2j1939(2060)
    37  #        data = [...., byte4, byte5, ...remaining 3 bytes...]
    38  def engine2j1939(rpm):
    39    # convert rpm to bits
    40    bits = math.floor(rpm / 0.125)
    41
    42    # convert bits to bytes
    43    byte4 = bits & 0xff
    44    byte5 = (bits >> 8) & 0xff
    45
    46    return byte4, byte5
```

Lines 29-33 Create the message arbitration ID

Lines 35-46 convert RPM into bytes required for engine speed message

Lab – /classroom/text_ic.py

```
> cat -n text_ic.py
     1	#!/usr/bin/python3
     2	# source: https://python-can.readthedocs.io/en/master/asyncio.html
     3	#   hereafter, [rtd] = source url through "master/"
     4	
     5	"""
     6	Using async IO with python-can to read a few interesting j1939 messages:
     7	    * ground speed (PGN FE49)
     8	    * engine speed (PGN F004)
     9	    * pto speed    (PGN FE43).
    10	
    11	And convert them to engineering units at write them to the command line.
    12	"""
```

Reference info and list of messages this tool can translate.

Lab – /classroom/text_ic.py

```
> cat -n text_ic.py

   127  async def main() -> None:
   ...
   141      # Create Notifier with an explicit loop to use for scheduling of callbacks
   142      #   notifier is used as a message distributor for a bus. Notifier
   143      #   creates a thread to read messages from the bus and distributes
   144      #   them to listeners. [rtd]/api.html#notifier
   145      loop0 = asyncio.get_running_loop()
   146      notifier0 = can.Notifier(canbus, listeners0, loop=loop0)
   147
   148      print("reading a few special J1939 messages from %s" % (channel))
   149      print(" ")
   150      print("use 'cansniffer' on %s to watch all traffic" %(channel))
   151      print("..........")
   152      print("^c to quit")
   153      while True:
   154          # Wait for next message from AsyncBufferedReader
   155          msg0 = await reader0.get_message()
   156
   157      # Clean-up
   158      notifier0.stop()
```

Async library allows us to be notified when the next message arrives

Lab – /classroom/text_ic.py

```
> cat -n text_ic.py

   85  def readcanbus(msg: can.Message) -> None:
   86      global mph, erpm, prpm
   87      """Regular callback function. Can also be a coroutine."""
   88      arbid = msg.arbitration_id
   89      data = msg.data
   90      priority, pgn, sa = arbid_decode(arbid)
   91      if pgn == 0xfe49:
   92         b1, b2, mph = pgnFE49(data)
   93      if pgn == 0xf004:
   94         b4, b5, erpm = pgnF004(data)
   95      if pgn == 0xfe43:
   96         b1, b2, prpm = pgnFE43(data)
   97
   98      #print("vcan0: %d, %x, %x : %x %x" %(priority, pgn, sa, b1, b2), end="\r")
   99      print("[%s] | " %(channel), end="")
  100      print("ground speed: %4.1f | " %(mph), end="")
  101      print("engine speed: %4.0f | " %(erpm), end="")
  102      print("pto speed: %4.0f | " %(prpm), end="\r")
  103
```

Lines 86-96 Handles messages, uses message specific functions to decode bytes into engineering units

Lines 98-102 write engineering units to the display

Lab – /classroom/text_ic.py

```
> cat -n text_ic.py

   68  # convert F004 to rpm  [engine speed]
   69  def pgnF004(data):
   70    b4 = data[3]
   71    b5 = data[4]
   72    bits = (b5 << 8) | b4
   73    rpm = bits * 0.125
   74    return b4, b5, rpm
```

Lines 68-74 Converts bytes from J1939 message into engineering units

Lab

- Assume “conflicting” messages are handled like in the Jeep