2020 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
CYBER TECHNICAL SESSION
AUGUST 11-13, 2020 - NOVI, MICHIGAN

SECURE HEAVY VEHICLE DIAGNOSTICS

Jeremy S. Daily, Ph.D., P.E.[1], Prakash Kulkarni, Ph.D.[2]

[1]Department of Systems Engineering, Colorado State University, Fort Collins, CO
[2]DG Technologies, Inc., Farmington Hills, MI

ABSTRACT

A cybersecurity exploit can be crafted to affect the vehicle diagnostic adapter system, which consists of the technician, vehicle diagnostic adapter, device drivers, and maintenance software all working together in a trusting relationship.

In this paper, application layer encryption of the SAE J1939 diagnostic traffic between the vehicle diagnostic application and the in-vehicle secure gateway is developed to mitigate the vulnerabilities in potential attack paths. The proposed encryption strategy uses AES-128, which uses 16-byte cipher blocks. The secure connection is established by adjusting the bit rate to over twice the normal speed and packing a single J1939 message into two encrypted sequential CAN frames,

The in-vehicle diagnostic gateway employs a hardware security module. A provisioning process is employed wherein the diagnostic application and the hardware security module both generate public-private key pairs. An elliptic curve Diffie-Hellman (ECDH) key exchange then takes place. Thus, each diagnostic session uses ephemeral symmetric session keys that are securely exchanged between the hardware security module and the diagnostics application.

This approach is effective in mitigating attacks originating at the driver (DLL) level, such as an attacker that would exfiltrate and modify data using the system and vehicle diagnostic subsystems in a Windows environment. Also, as the secure key system can be centrally administered, the ability for user attribution through key management is possible.

While the approach requires the addition of a hardware security module on the vehicle, the hardware strategy presented could be implemented in an arbitrary electronic control module on the vehicle. Vulnerabilities and mitigations are explained in detail to provide a solution to secure diagnostic sessions for heavy vehicles.

Citation: J. Daily, P. Kulkarni, "Secure Heavy Vehicle Diagnostics", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 13-15, 2020.

1. Introduction

Heavy vehicles undergo routine maintenance and inspections, which entails connecting a vehicle diagnostic adapter (VDA) to the diagnostic port on the vehicle and running some diagnostic software on the shop computer. This process happens frequently and represents a potential cybersecurity vulnerability. While the individual vehicle is often disconnected from the diagnostic session, the computer, usually a PC laptop running Windows, is frequently connecting to the Internet. In fact, many diagnostic software systems require Internet connections for licensing and firmware updates.

The diagnostic connections for heavy vehicles typically follow the recommended practice (RP) number 1210 as published by the Technology and Maintenance Council (TMC) of the American Trucking Association (ATA) [1]. The concept behind the RP1210 approach was to decouple the diagnostic software from the vehicle network. This gives the vehicle maintainer the ability to use only one RP1210 compliant device that will work with the different diagnostic software needed for the subsystems of a heavy vehicle. The programmers and developers of the maintenance software can write to the RP1210 API and the vendors of the different vehicle diagnostic adapters (VDAs) can serve the API through their vendor specific dynamically loaded libraries (DLLs).

From a security perspective, this approach implies a sense of trust on behalf of the diagnostic software regarding the sanctity of the vehicle network traffic. This trust is only maintained if the VDA system faithfully sends and receives the actual vehicle network traffic. However, there are no controls in place for the maintenance software or ECU for validating the integrity and authenticity of the data passing through the RP1210 subsystem.

Cybersecurity associated with heavy vehicles have recently become an issue. While cybersecurity related to passenger cars was brought to the forefront in 2010 by Kosher, et al. [2], the heavy vehicle industry lagged a few years. The National Motor Freight Traffic Association, Inc's whitepaper on the state of heavy vehicle cybersecurity in 2015 [3] marks a beginning of the industry efforts to address cybersecurity efforts. This provides a decent survey of the literature and cybersecurity efforts taken place. The heavy vehicle industry became more aware of the issue as it started standard making efforts to address cybersecurity. Also, descriptions of building cybersecurity centric testbeds were presented in the literature [4]. Product briefs and whitepapers have come out regarding implementations for technologies around diagnostics for passenger cars [5], but there are no cybersecurity studies in the literature for RP1210 based diagnostic systems.

2. Assumptions

The context of this work is based on some enabling assumptions. These assumptions are based on author experiences with commercial heavy vehicles with limited knowledge of military or proprietary systems.

We also assume the reader is familiar with some basic cryptography primitives and cybersecurity concepts. Ross Anderson provides a decent introduction into modern cybersecurity [6].

Vehicle System Architecture

We assume all relevant diagnostic communication takes place over a single CAN channel as defined in SAE J1939. This CAN channel is typically found on pins C and D of the Deutsch 9-pin diagnostic connector in the driver compartment of the cab of the vehicle. The bit rate for the CAN bus is set to 250,000 bits per second.

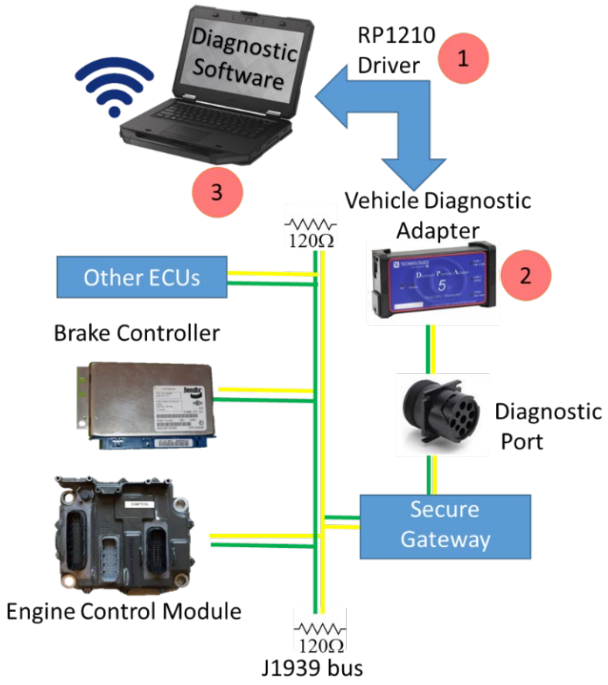


Figure 1: Vehicle architecture showing potential attack vectors in the diagnostic system.

We assume there are no other diagnostic communications, like a J1708/J1587 network or an installed telematics device.

As shown in Figure 1, the secure gateway separates the main SAE J1939 backbone from the diagnostic port. There are no communications through the diagnostic port for anything other than vehicle diagnostics.

Physical access to the vehicle is required to connect a vehicle diagnostics adapter to the diagnostic port. Once the VDA is connected, the vehicle system adds the additional hardware and software stack of the diagnostic services, to include the back-end servers.

Attack Vectors

For this work, we are assuming an attacker will try to penetrate the Windows based diagnostic computer or the vehicle diagnostic adapter. We assume the diagnostic application is secure and protected. Protecting the diagnostics software against a cybersecurity attack may be important, but it is beyond the scope of this paper. The security of applications running on a PC would employ a different set of solutions and can benefit from existing strategies.

We also assume the SAE J1939 network traffic is safe. This means the secure gateway will faithfully report the J1939 traffic it sees to the PC Application. While attacks to the J1939 network are possible, they often require physical access or tampering with the supply chain. Internal J1939 intrusion defenses are a subject of continuing study but are beyond the scope of this paper.

An adversary can attack the diagnostic systems in the following ways:

To summarize, the proposed solution aims to mitigate attacks anywhere in the communication chain from the PC based diagnostics application to the diagnostics gateway.

3. Requirements

The goal of securing heavy vehicle diagnostics needs to be broken down into smaller measurable requirements by which the overall system can be evaluated. These requirements drive design and implementation decisions. There are many goals and requirements to consider, so the following subsection break them down by their scope.

System Requirements

Enumerated system requirements (SRs) for the vehicle and diagnostic system are as follows:

SR1: Maintain compatibility with existing J1939 Architectures

SR2: Provide a solution that is agnostic of the vehicle diagnostic adapter.

SR3: Enable offline diagnostics sessions.

SR4: Store CAN Data Logs based on event triggers.

Cybersecurity Requirements

The cybersecurity requirements (CRs) are enumerated as follows:

CR1: Use unique key material so any key leakage does not compromise other systems.

CR2: Use secure storage hardware for private key storage on the vehicle.

CR3: Use existing best practices for cryptographic implementations. For example, AES-128 can be used for symmetric encryption, Elliptic Curve Cryptography can be used for asymmetric encryption. Any new or untested encryption systems shall not be used.

CR4: Any sensitive key material should be encrypted for storage.

Design Alternatives

Before explaining the prototyped solution in detail, this section will share some approaches that were considered, but ultimately dismissed.

Alternative 1: Use a software only solution. This approach would improve the capabilities of each electronic control module to support secure diagnostic communication. This approach was abandoned because the authors do not have access to the source codes to run on the individual modules on the vehicle.

Alternative 2: Use an alternative communication path that circumvents the RP1210 communication system. Perhaps Ethernet or wireless technologies could be used This path would add another hardware device to the solution and disrupt a well-known and accepted workflow of connecting a vehicle diagnostic adapter to the diagnostic port.

Alternative 3: Secure each individual path of communications uniquely within the RP1210 chain. This approach would not transfer across the different vendors of vehicle diagnostic adapters, which was one of the issues that led to the creation of RP1210 over 30 years ago.

The approach taken is hardware and driver agnostic when it comes to the RP1210 vehicle diagnostic adapter.

4. Hardware Prototype

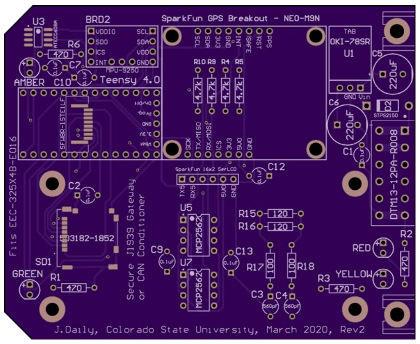


Figure 2: Printed Circuit Board for hand assembly

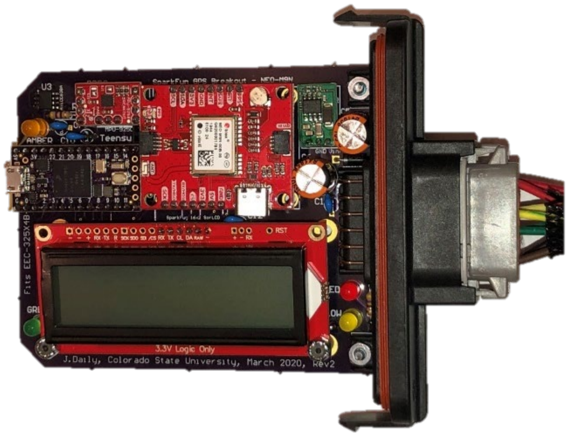


Figure 3: Assembled circuit card assembly

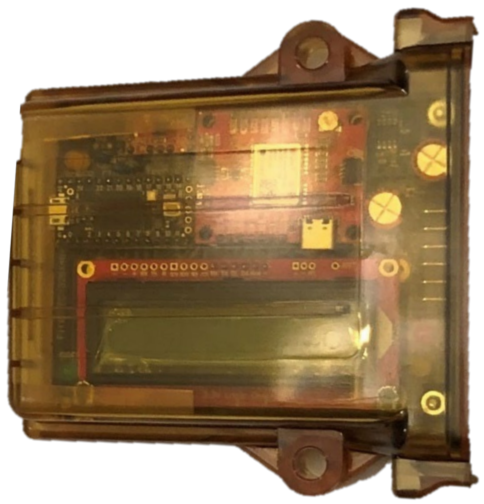


Figure 4: Ruggedized enclosure to mount in a vehicle

In this section, we provide details of the hardware prototype with enough detail for the reader to reproduce our example. The following are the essential components for the hardware design:

The solution to secure the diagnostics communication requires a platform to perform the security algorithms. Since access to source codes and tool chains for existing gateways or electronic control units is not practical, we designed our own gateway leveraging open source software and evaluation hardware. The platform of choice was the Teensy 4.0 evaluation board from PJRC.com. This evaluation board features a 600MHz iMXRT1062 ARM-32bit processor with three CAN channels, two of which are available on the 0.100" spaced headers of the Teensy 4.0.

The essential components of the circuit design include the Teensy 4.0, both CAN transceivers, the terminating resistors, and the ATECC608A hardware security module. The SD Card provides the additional capability for logging data.

The display, GPS, and inertial measurement unit add features for status updates and contextual information. These additional features are not needed for secure diagnostic communication.

The complete schematic diagram for the Secure Gateway Prototype is shown in the Appendix. The Teensy 4.0 evaluation board is connected to the CAN busses with two Microchip MCP2562 CAN Transceivers, which forms the break in the line for the diagnostics communication. The printed circuit board (PCB) was laid out for through-hole parts and is available to purchase from Osh Park. The PCB fits in a Deutsch/TE EEC-325X4B enclosure designed for automotive use. The enclosure is sealed with the Deutsch DTM13-12PA-R008 PCB to connector system. Admittedly, the space in the enclosure is tight when using the additional prototyping features and display. Also, the SD Card holder, FFC ribbon cable, and the ATECC608A security module are surface mount packages, which will require additional skill to hand assemble the boards.

Figure 2 shows the layout of the printed circuit board used in this study. The additional silkscreen lettering helps with part placement and assembly by hand. The completed system is shown in Figure 3 and the enclosed assembly is shown in Figure 4.

Installing the Secure Gateway

The Gateway enclosure can be mounted at any convenient location on the vehicle.

The wired connector splits the connection to the diagnostic

port. The wiring harness is for the Secure Gateway is shown in Figure 5. The harness shows two 9-pin diagnostic connectors; the plug with female contacts is on the left of the illustration and the socket with male contacts are on the right. The typical heavy vehicle with have the Diagnostic Network exposed in the driver's compartment with the male pins. The vehicle diagnostic adapter will connect using the housing with female contact, as shown on the left of Figure 5. This means the gateway can be installed into the existing diagnostic port and the new port installed in its place. This installation creates a brand new point-to-point CAN bus for use only by the vehicle diagnostics adapter and the secure gateway. Since this new network is not connected directly to the J1939 network, we can define its message structure and bitrate.

5. Secure Diagnostics

The strategy for maintaining confidentiality of the messages passing through the VDA, its drivers, and the USB stack is to encrypt each CAN frame using AES-128. However, 128 bit block ciphers require 16 byte blocks and CAN only transmits 8 bytes (64-

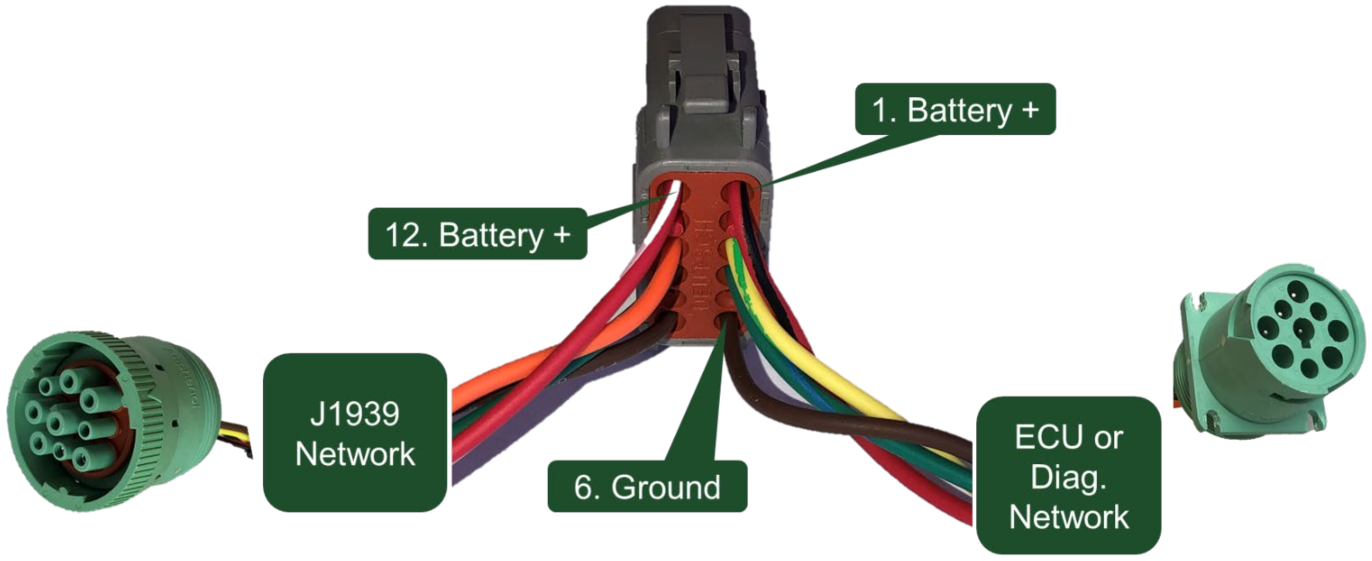


Figure 5: Wiring harness for the Secure Gateway

bits) at a time. This means we will need to take each J1939 frame and pack it into two high speed encrypted CAN frames. These packed frames will be enciphered and send over the diagnostic CAN network in ordered pairs. The PC Application will read these raw 11-bit CAN frames and reassemble the message pairs, check their validity, and present the decrypted J1939 message to the diagnostic application for processing.

The messages on the newly created point-to-point diagnostic CAN Bus use 11-bit CAN IDs at 1,000,000 bits per second. Since this is twice the speed of the fastest J1939 implementation (500kbps), the diagnostic CAN Bus should be able to keep pace with the original J1939 network, even if each J1939 message was expanded and fit into 2 messages on the fast diagnostic CAN.

In this scheme, the first three bits of the 11-bit CAN IDs are used to determine the message type. There are three types of messages used, presented below as C header defines:

```
#define SESSION_CONTROL 0x100
#define ENCRYPTED_DATA  0x200
#define HEART_BEAT      0x400
```

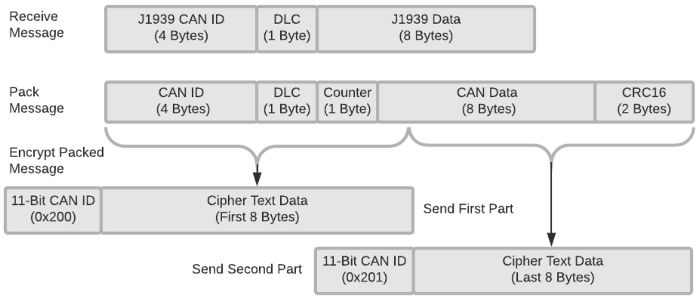


Figure 6: Separating a single J1939 CAN Frame into two 8-byte

To send encrypted data, the message ID starts with a 0x200. The sequence is determined by the first 8 bytes. Even values of the ID contain the first part of the block cipher and the odd values of the ID contain the last 8 bytes of the block cipher. A breakdown of mapping a J1939 message to the pair

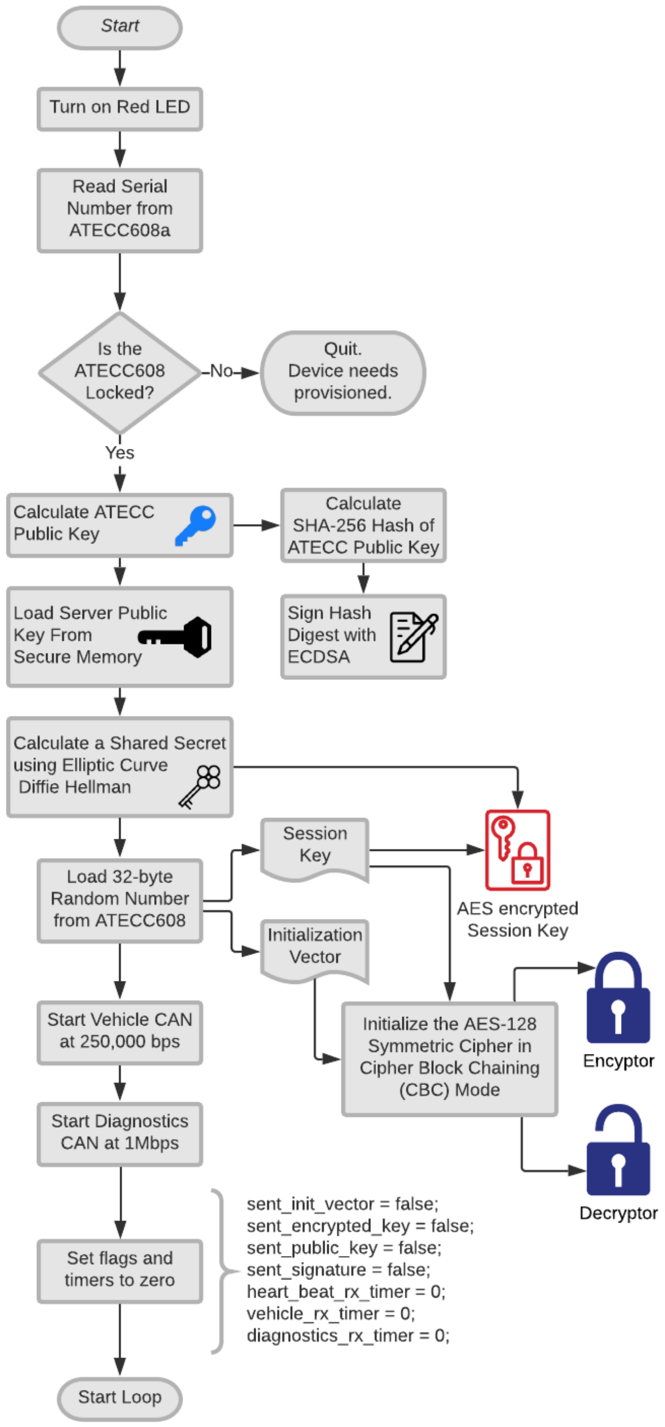

Start
Turn on Red LED
Read Serial Number from ATECC608a
Is the ATECC608 Locked?
No
Quit. Device needs provisioned.
Yes
Calculate ATECC Public Key
Calculate SHA-256 Hash of ATECC Public Key
Load Server Public Key From Secure Memory
Sign Hash Digest with ECDSA
Calculate a Shared Secret using Elliptic Curve Diffie Hellman
Session Key
AES encrypted Session Key
Load 32-byte Random Number from ATECC608
Initialization Vector
Start Vehicle CAN at 250,000 bps
Initialize the AES-128 Symmetric Cipher in Cipher Block Chaining (CBC) Mode
Encyptor
Start Diagnostics CAN at 1Mbps
Decryptor
Set flags and timers to zero
sent_init_vector = false;
sent_encrypted_key = false;
sent_public_key = false;
sent_signature = false;
heart_beat_rx_timer = 0;
vehicle_rx_timer = 0;
diagnostics_rx_timer = 0;
Start Loop

Figure 7: Flow chart describing the operations used to startup the Secure Gateway

of ENCRYPTED_DATA messages is shown in Figure 6. The cipher text data is calculated using the AES-128 algorithm in cipher-block-chaining mode (AES-CBC).

Implementation Flow Chart

There are four main routines to describe the software operation of the Secure Gateway. There are two standard routines, startup and loop, followed by two interrupt routines to handle the reception of CAN frames. There are two CAN frame message processing routines: one for receiving diagnostic communications from the VDA and the other for processing messages from the J1939 vehicle network.

The flow chart in shows the startup routine. The first operation communicates with the ATECC608A security module to determine its serial number and state. The serial number will be used by the diagnostics software on the PC to determine what key is calculated to share a secret. For the ATECC608A to perform the necessary calculations, the configuration and data zones need to be locked. A generated private key should be stored and locked in the ATECC608A from which it can calculate a public version of that key.

Once the public version of the Secure Device Key is computed, in can be loaded into the microprocessor memory and shared upon request. To ensure against man-in-the-middle attacks, the public key is digitally signed, so the receiving node can have high confidence the public key is from the correct device.

The ATECC608A also claims to have a true random number generator (TRNG) that can produce 32 bytes at a time. The startup procedure requests these bytes and splits them into two 16-byte words; one for the initialization vector and the other for the AES-128 session key. From these random numbers used once (nonce) values, an encryptor and decryptor are setup using the AES-128 algorithm in CBC mode.

With a symmetric encryptor and decryptor setup on the Secure Gateway, there needs to be a way of sharing the initialization vector and session key with the Secure PC Diagnostics Application (PC App). The method to share the session key is to send an encrypted version of the key from the Secure Gateway to the PC App. The session key is encrypted with a computed shared secret derived

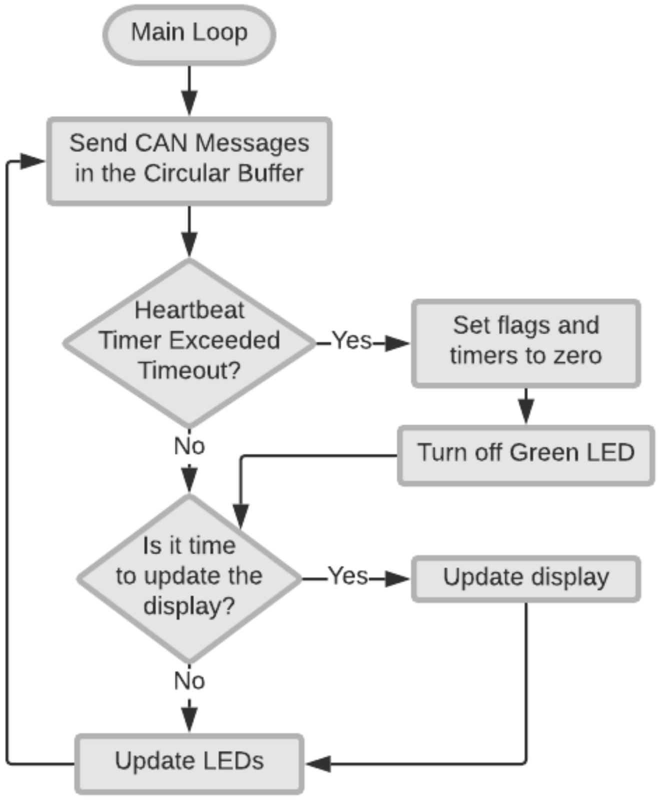

Main Loop
Send CAN Messages in the Circular Buffer
Heartbeat Timer Exceeded Timeout?
Yes
Set flags and timers to zero
Turn off Green LED
No
Is it time to update the display?
Yes
Update display
No
Update LEDs

Figure 8: Main loop with basic utility calls

from the Elliptic Curve Diffie-Hellman key exchange routine. However, the PC App does not request the public key from the Secure Gateway. Instead, it requests the key from the provisioning service based on proper credentials. This provides an opportunity for usage attribution and user accountability, which is an improvement in the cybersecurity posture compared to current approaches.

The encryptor is setup to convert clear J1939 based vehicle CAN frames in enciphered diagnostic CAN frames. The decryptor is setup for the opposite operation, which is to take encrypted diagnostics CAN message pairs and convert them to J1939 vehicle network CAN frames. The same ephemeral session key and initialization vector is used for both the encryptor and decryptor.

Once the key material is initialized, the CAN busses are started, one at the speed of the vehicle bus and the other at the fastest rate specified for CAN 2.0, which is 1Mbps. Once the CAN message receive interrupt handlers are registered, the system finishes with the startup routine and begins the main processing loop.

The main loop, shown in Figure 8, is predominantly responsible for updating the displays and calling the service routine that sends the CAN messages. The CAN messages are written to a circular buffer first, then a service function is called to push the CAN message from the circular buffer to the CAN Controller.

Each time a CAN message is received from the vehicle CAN bus, which is the J1939 network, an interrupt service routine is executed. This routine is shown in Figure 9. The first section of the receive routine is to check the flags setup for a successful session. The session is determined from a successful key exchange and a consistent heartbeat signal. If those prerequisites are confirmed, then the message can be enciphered with the encryptor object and transmitted as a pair of CAN frames.

The heartbeat signal is a CAN frame from the PC App that is designated by starting the 11-bit CAN-ID with 0x400. This message is crafted by the PC App thread responsible for secure communication. When a Secure Gateway decrypts the message, it resets a timer and continues with the secure diagnostics session. To inform the PC App that the Secure Gateway is still connected, it reflects the message. Since these messages are encrypted, the session key and sequence must be preserved to have a legible decoding of the heartbeat.

The decrypted message contained in the heartbeat is a J1939-like frame with a 4-byte ID field representing a counter, a data length code (DLC) of 8, and an arbitrary message of 8 bytes. This so-called CAN frame is packed, encrypted and sent according to the message structure shown in Figure 7. Since the heartbeat contains a 16-bit cyclic-redundancy check, if the encrypted message was either manipulated or decrypted with the wrong cipher, the CRC check would fail, and the session would timeout.

When the PC App is not running, the Secure Gateway is not receiving any heartbeat signals and the session has not been established. The Secure Gateway is holding onto a session key from the startup sequence and it is ready to share upon request. To initiate the key exchange, the Secure Gateway performs the interrupt service routine shown in Figure 10.

The routine checks for three different kinds of CAN message IDs as defined in Section 1.5. If the first nibble of the 11-bit ID is of the SESSION type, the following options are defined:

```
#define REQUEST_PUBLIC_KEY          0x110
#define SEND_PUBLIC_KEY_DATA        0x120
#define REQUEST_ENCRYPTED_KEY       0x130
#define SEND_ENCRYPTED_KEY          0x140
#define REQUEST_INIT_VECTOR         0x150
#define SEND_INIT_VECTOR            0x160
#define REQUEST_SERIAL_NUMBER       0x170
#define SEND_SERIAL_NUMBER          0x180
#define REQUEST_PASSCODE            0x190
#define SEND_PASSCODE               0x1A0
#define RESET_SESSION               0x1B0
#define REQUEST_SIGNATURE           0x1C0
#define SEND_SIGNATURE              0x1D0
#define SESSION_ERROR               0x1E0
#define SESSION_ABORT               0x1F0
```

Notice the special case of 0x1F0 is an abort message that would restart the Gateway and develop new session keys.

Since the PC App initiates the communication, it first requests the serial number of the secure gateway. The define for this request is 0x170. This means that if the Secure Gateway sees a message with the ID of 0x170, then it will reply with the 11bit ID of up to 16 messages of 0x180 + n, where n = 0x0,1,2, …, F, is the sequence number.

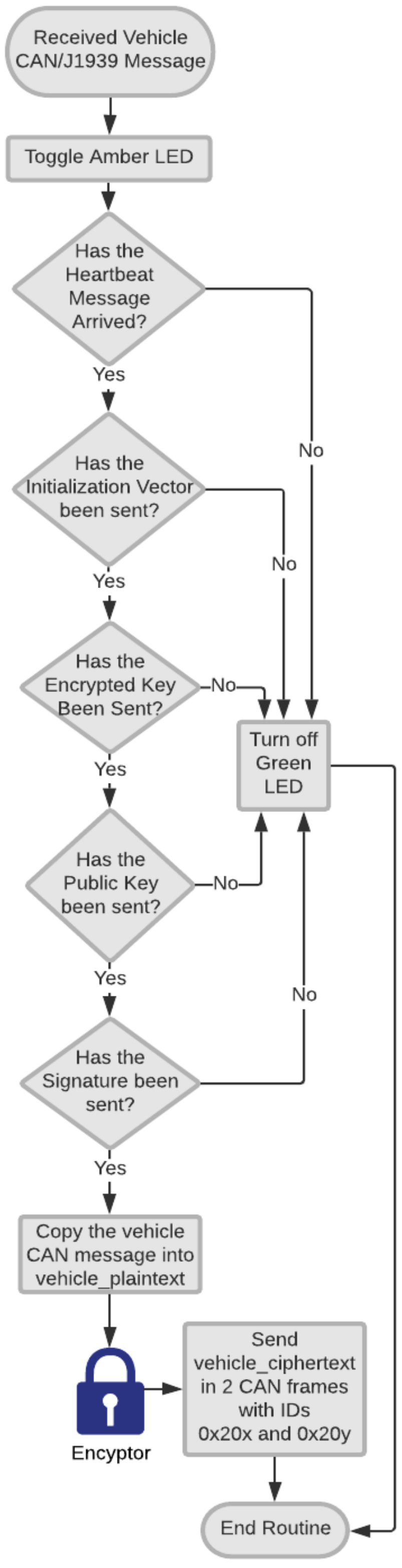

Received Vehicle CAN/J1939 Message
Toggle Amber LED
Has the Heartbeat Message Arrived?
Yes
No
Has the Initialization Vector been sent?
Yes
No
Has the Encrypted Key Been Sent?
No
Yes
Turn off Green LED
Has the Public Key been sent?
No
Yes
No
Has the Signature been sent?
Yes
Copy the vehicle CAN message into vehicle_plaintext
Encyptor
Send vehicle_ciphertext in 2 CAN frames with IDs 0x20x and 0x20y
End Routine

Figure 9: Vehicle Message Interrupt Service Routine

Since the largest data exchange using ECC with the P256 curve is 64 bytes, only 8 messages maximum are needed to respond with the requested data.

After the PC App requests and receives the serial number of the Secure Gateway, it needs to calculate the same shared secret. The shared secrets are calculated using their own private key and the public key of the other device. The public key used by the Secure Gateway was loaded from secure memory upon boot and subsequently used to determine a shared secret, the PC Application needs to calculate its shared secret based on the private key that the Secure Gateway already had. This means the PC Application needs to either 1) request that the session key be decoded from a secure service, or 2) check out the private key matching the pre-shared public key so it can compute its own shared secret. To understand how a key checkout or secret calculation is implemented, we need to discuss device provisioning.

6. Key Management

The strategy to secure diagnostic communications is to always use ephemeral session keys generated by a high-quality random number generator built into the Secure Gateway.

Secure Device Provisioning

When the Secure Gateway is ready for final testing and assembly, it needs to be provisioned. At a minimum, the ATECC608A is configured such that a self-generated P256 ECC key is generated and locked into a slot. In this prototype, we designated Slot 0 to contain this unique private key. This key should never be readable or able to be leaked. From the key stored in Slot 0, a public key can be generated and shared with a database. The database key is the 72-bit unique ID in ASCII form (18 characters). In addition to storing the public key from the ATECC608A chip in the database, a server private key from the P256 curve is generated along with its corresponding public key.

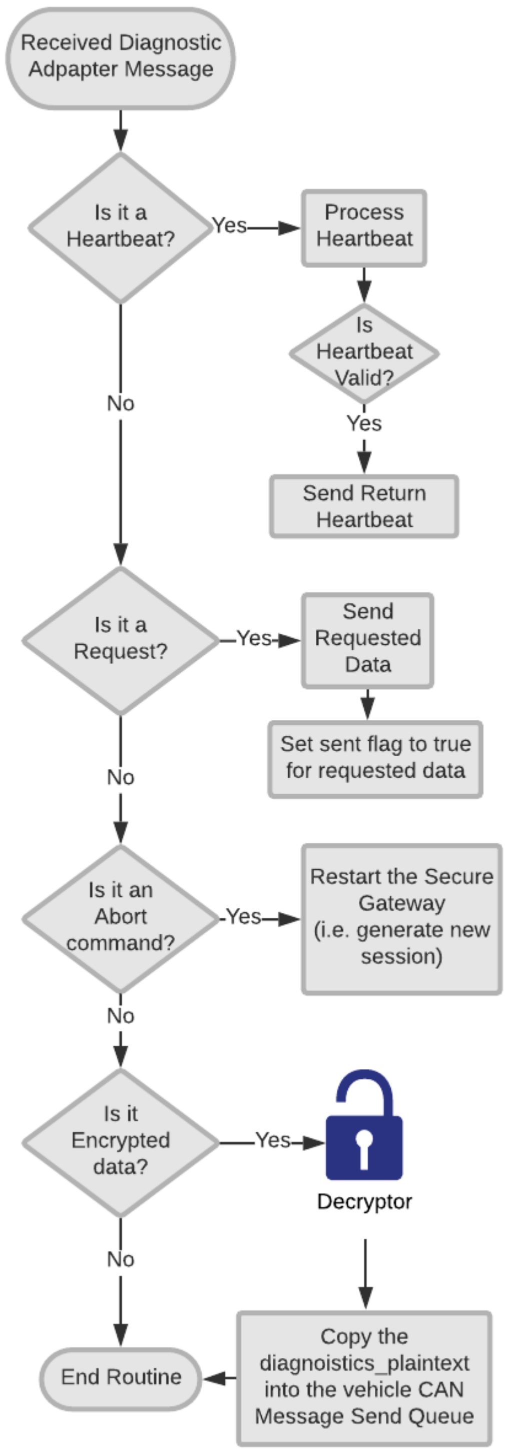

Received Diagnostic Adpapter Message
Is it a Heartbeat?
Yes
Process Heartbeat
Is Heartbeat Valid?
Yes
Send Return Heartbeat
No
Is it a Request?
Yes
Send Requested Data
Set sent flag to true for requested data
No
Is it an Abort command?
Yes
Restart the Secure Gateway (i.e. generate new session)
No
Is it Encrypted data?
Yes
Decryptor
No
End Routine
Copy the diagnoistics_plaintext into the vehicle CAN Message Send Queue

Figure 10: Flow diagram to process CAN frames from the Vehicle Diagnostic Adapter and the PC Application

This process generates the ability for a pre-shared secret to be calculated. Furthermore, the ATECC608A can update the server public key using a privileged write feature.

All key material should be encrypted at rest with a key tied to a hardware security module. In the prototype example, we have implemented the database as Amazon Web Services DynamoDB and each device has a data key produced and encrypted using the AWS Key Management Service. This provides an audit trail into who and when the key material is being used.

By pre-sharing public keys, it is fairly straightforward to verify if requests are from legitimate sources. If the public keys do not match, then the device is not part of the ecosystem.

Checking Out Keys

There may be cases where a maintainer may want to initiate a secure diagnostic session when access to the server and database is not feasible. In this case, a set of anticipated keys can be checked out from the server and stored locally on the PC App machine. At first, this may sound very risky since the device private keys will be on a potentially unsecured computer. To mitigate this risk, the private keys checked out from the server are in the encrypted PEM format. Furthermore, the passcode to decrypt the PEM key is derived from the pre-shared secret and a passcode. This means the operator needs to have physical access to the vehicle and the gateway to decrypt the PEM key in addition to an 8-byte unique passcode. Having access to the vehicle without an authorized user passcode or having a passcode without the hardware renders the key material meaningless.

During the provisioning process, the preshared secret is calculated on the server because the server has generated a private key and shared the public key. The ATECC608A has done the same. Two random strings of characters are generated on the server. The first string is used as a passcode. The other string is padded with 8 zeros to make a 16-byte block. This 16-byte block is encrypted using the AES-128 in ECB mode to create a ciphertext of the PEM key pass code.

Creating the PEM file key

The implementation used during provisioning is an AWS lambda function written in Python. The source code is available as open source on Github[1]. The algorithm of the provisioning is as follows:

If an authorized, valid user requests a key for offline use, then the server will respond with the Encrypted Server Private Key and the Random device code. The Random device password will be emailed to the authorized user. Since the only way to derive the actual password is through the AES-128 cipher based on the preshared key, only the server or the ATECC608A can compute the passcode. This means there are two levels of security for the unique key material: the passcode (something you know) and the device code (something you have).

Secure Session Key Exchange

The PC App initiates the session. In the Python -based prototype, the SecureMessageThread class was defined to handle the setup, encryption, and decryption of the PC App. The thread communicates with the rest of the program through queues and requires function prototypes to communicate with the VDA DLL.

To demonstrate the setup when starting the SecureMessageThread in the PC App, a transcript of the network traffic is shown in the following paragraphs. The CAN traffic is the direct output of the Linux SocketCAN can-utils program called candump.

```
can1  1F0   [8]  00 00 00 00 00 00 00 00
             can1  170   [0]
can1  180   [8]  01 23 71 6D B2 52 F4 6C
can1  181   [8]  EE 01 01 01 56 6C 4A 4B
             can1  150   [0]
can1  160   [8]  F6 73 9D 2A 5E 03 B0 CB
can1  161   [8]  62 FE 82 72 84 41 55 DA
             can1  130   [0]
can1  140   [8]  6F 02 7E E7 AA 83 1B 06
can1  141   [8]  07 68 CD A9 43 5D 0B 82
             can1  110   [0]
```

The Secure Gateway sends the device public key

```
can1  120   [8]  56 6C 4A 4B D2 40 82 F9
can1  121   [8]  85 42 51 64 7B 97 25 35
can1  122   [8]  49 A4 1D 53 69 92 D3 6D
can1  123   [8]  14 0F B7 A6 6A CA 94 C9
can1  124   [8]  81 FF B5 A3 8F 5C D9 BF
can1  125   [8]  9E 20 46 CE 5B 77 72 F9
can1  126   [8]  1F BA BA 7A 88 90 BB 45
can1  127   [8]  51 E1 83 A5 B3 03 9F 28
             can1  1C0   [0]
can1  1D0   [8]  1D 95 7D D4 B0 4B BB CE
can1  1D1   [8]  FA 76 23 AB 6C CB AF F6
can1  1D2   [8]  F7 D8 42 E5 35 A2 60 98
can1  1D3   [8]  40 6C 20 E9 E3 6B EA FE
can1  1D4   [8]  B4 49 9E D8 93 F3 D4 78
can1  1D5   [8]  86 20 80 EC AA 64 FF 3A
can1  1D6   [8]  64 7E 54 91 86 AD 60 E0
can1  1D7   [8]  0B 25 53 3E 92 1B 09 9D
```

The PC App can now verify the public key.

```
can1  190   [8]  42 41 74 6B 75 39 38 6A
can1  1A0   [8]  98 E6 60 B6 17 A3 7C 65
can1  1A1   [8]  39 9C 4A 50 8D CB 4C A6
```

The PC App prepends the passcode to the returned device code to create a 24-byte key for the encrypted PEM file on the PC. This PEM key is loaded into memory and used to compute the shared secret that was used to encrypt the session key. Once the session key is decrypted and the encryptor and decryptor are initialized, a heartbeat is transmitted.

```
can1  400   [8]  BA 21 FA 3E A6 E5 7E 95
can1  401   [8]  D6 DD 63 42 04 20 04 2F
can1  406   [8]  95 E1 9E D9 32 75 DE 55
can1  407   [8]  09 DC 09 4D B8 3C DF CF
can1  200   [8]  C9 DA F9 C2 E1 CC A3 BC
can1  201   [8]  3D 04 50 D2 49 9D A2 E0
can1  202   [8]  41 83 BF E1 91 26 89 46
can1  203   [8]  50 D7 A2 18 FE A6 1C B6
can1  204   [8]  FC C6 DC DB 87 6F 8B CC
can1  205   [8]  E7 C8 10 A2 AD E2 6D 8F
```

[1] https://github.com/SystemsCyber/CANWatermarking/blob/master/serverless/provision.py

```
can1  206   [8]  E6 A3 61 28 29 12 CD 7C
can1  207   [8]  42 5A 5F 26 F0 1C 7D 58
can1  208   [8]  2E 1A FA 8E 63 2B DF C7
can1  209   [8]  AA BB 06 C1 6E 1F 6A C9
```

Analysis of the Session Setup

The transcript of the session setup shows about 200 bytes of data are exchanged. However, the public key and its signature may not be needed, except for provisioning. Since the ATECC608A secret key is not changing and the server public key is not changing, the derived shared secret is not changing. Therefore, it can be securely stored in one of the protected slots of the ATECC chip and the need to exchange public keys is eliminated.

Key material on the CAN bus is either AES encrypted, as in the session key, or is only a partial key, as in the device code. The need to send the device code over the CAN bus is because the PC App cannot reach the backend server. With a connected PC App, there would be no need for the device code because the encrypted session key would be decrypted on the server and transmitted back to the PC App using an https protocol (TLS).

The benefits of this setup approach include

If the requirement to be able to have diagnostic sessions without Internet connectivity is relaxed, then the encrypted session key could be decrypted on the server with only the serial number as the accompanying piece of information.

7. Cybersecurity Attacks

To test the Secure Gateway and PC App functionality and demonstrate the efficacy of the approach a few cyberattacks were implemented on a test bench. These attacks focused on two entry points: 1) the DLL driver and 2) the VDA firmware. The attacks targeted two types of messages: A) single frame J1939 messages, 2) Multiframe messages using the J1939 Transport Protocol. Other transport protocol attacks would be similar.

The attacks manifest themselves as man-in-the-middle attacks. The effect would be the same if the attack was implemented as a Shim DLL, a rogue VDA, or an actual CAN man-in-the-middle device.

Shim DLL

The Shim DLL pretends to be an actual RP1210 compliant device driver with legitimate function calls to perform RP1210 tasks. However, instead of having an actual device driver to implement the commands and send messages, the Shim DLL loads an actual vendor supplied DLL and passes through the function calls. However, the Shim DLL has the ability to change the data before it presents the message to the PC App or the actual VDA driver. Hence, the Shim DLL acts as a man-in-the-middle.

Compromised VDA

A compromised VDA would have a version of firmware that does not faithfully report the message traffic between the vehicle networks and the PC. Creating an attacker VDA requires some knowledge of the device driver and device firmware and access to the tool chain. However, update mechanisms exist where an attacker could convert a VDA to an attack node.

An example of a purpose built compromised VDA is shown in Figure 11. Obviously, the overt graphics in Figure 11 would not be present in a realistic setting. Many RP1210 vehicle diagnostic adapters have no cybersecurity protections and should not be trusted.

Figure 11: A compromised vehicle diagnostics adapter (VDA)

Single Frame Attacks

A single frame attack on a CAN bus with a man-in-the-middle follows a basic algorithm:

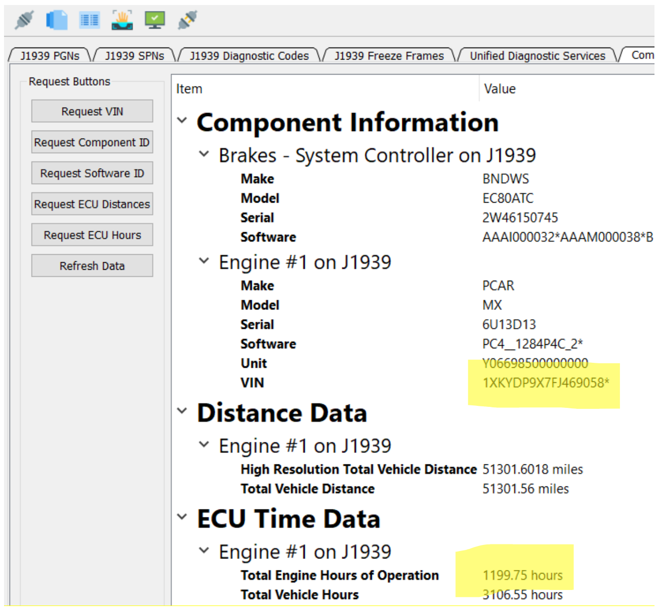
An example attack is to affect the Engine hours report. Many rental companies use the Engine hours as a method to bill the customer for use of machines. If the diagnostic system underreported the hours, then the customer may need to pay less. Total Engine Hours of Operation are in the first four bytes of the J1939 message with a PGN of 65253.

Multi Frame Attacks

SAE J1939 networks have a Transport Protocol as define in SAEJ1939-21 where the parameters that are over 8 bytes are broken into CAN frames and then reassembled to present the original data. The challenge for attacking this data is the PGN to identify the data type is contained in the Transport Layer – Connection Management (TP-CM)_ as the last three bytes in the data field. If an attacker wanted to change the VIN as it passed from the J1939 network to the diagnostics application, the algorithm would have to look for the TP-CM message to identify the PGN of 65260. Since the PGN is encoded in reverse byte order, the TP-CM message ending in 0xEC, 0xFE, 0X00 would be the setup message for transferring the VIN. Subsequent Transport Protocol- Data Transfer (TP-DT) messages, with a PGN of 60160, will contain the data for the VIN. These messages are the ones for manipulation.

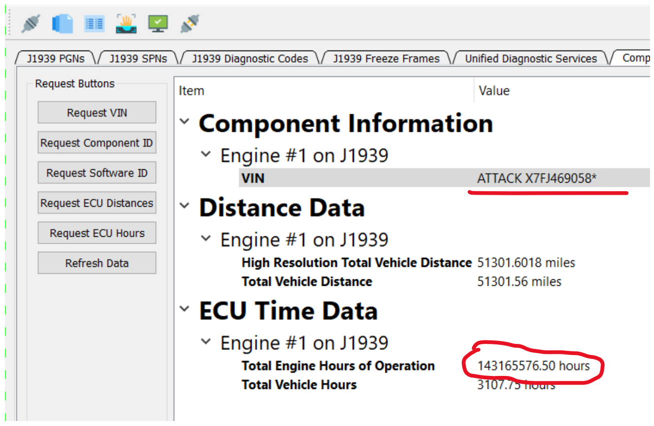
Attack Example

An example of manipulating the Engine hours and VIN is show in Figure 11 and Figure 12. The valid data, as reported on the J1939 network, is shown highlighted in yellow. After an attack was launched, the data was changed as shown in Figure 12. The attack on the Engine hours was to change all the bytes to 0xAA. The 32-bit unsigned integer 0xAAAAAAAA is 2,863,311,530 in decimal. The conversion factor for engine hours according to J1939 is 0.05 hours per bit, which gives the displayed value of 143,165,576.5 hours in Figure 12. Changing the first part of the VIN to ATTACK should be readily seen in the figures.



Item	Value
Component Information	
Brakes - System Controller on J1939	
Make	BNDWS
Model	EC80ATC
Serial	2W46150745
Software	AAAI000032*AAAM000038*B
Engine #1 on J1939	
Make	PCAR
Model	MX
Serial	6U13D13
Software	PC4_1284P4C_2*
Unit	Y06698500000000
VIN	1XKYDP9X7FJ469058*
Distance Data	
Engine #1 on J1939	
High Resolution Total Vehicle Distance	51301.6018 miles
Total Vehicle Distance	51301.56 miles
ECU Time Data	
Engine #1 on J1939	
Total Engine Hours of Operation	1199.75 hours
Total Vehicle Hours	3106.55 hours

Figure 12: Diagnostic software displaying the normal VIN and Engine hour data.



Item	Value
Component Information	
Engine #1 on J1939	
VIN	ATTACK X7FJ469058*
Distance Data	
Engine #1 on J1939	
High Resolution Total Vehicle Distance	51301.6018 miles
Total Vehicle Distance	51301.56 miles
ECU Time Data	
Engine #1 on J1939	
Total Engine Hours of Operation	143165576.50 hours
Total Vehicle Hours	3107.75 hours

Figure 13: Results of an attack through the diagnostics system that show different hours and VIN.

Exfiltration Attacks

The RP1210 system is capable of logging all the data through the vendor specific DLL. This logging feature can be enabled in the Windows environment without the need for any additional hardware. If a diagnostic computer is compromised, logging feature of the Shim DLL could be enabled, and the Shim DLL could transfer that data to an unauthorized organization. This process of exfiltration can lead to a significant amount of readiness and operational data for heavy vehicles. Corporations could assess their competition with greater intelligence (i.e. determine routes and customers), and defense operations may not want other nation states to see vehicle utilization data. Instead of eliminating leakage of data from the Windows machine, we assume data may be leaked. If this is the case, we want the data to mean nothing to the recipient.

Mitigating the Attacks

Since the attacks rely on identifying the messages coming through the diagnostic services, the core trigger for implementing the attack is not present due to the encrypted messages having no identifiable pattern. This means an attacker could only randomly affect messages. If the messages were manipulated between the Secure Gateway and the PC application, the decryption would fail and the CRC would not match. The message would be dropped.

If data were exfiltrated, the AES-CBC data stream would be meaningless without the key. Since the session key is securely exchanged using ECDH, the likelihood of any actionable intelligence from this data leakage is minimal.

8. Concluding Remarks

The diagnostics and maintenance system of J1939 enabled heavy vehicles uses a potentially vulnerable vendor agnostic system known as RP1210. The RP1210 system came to being from market pressures to homogenize tooling and adapters needed to communicate with heavy vehicles in the 1990s, well before the ubiquitous connectivity of today. The RP1210 system in use today uses third party vehicle diagnostic adapters and device driver software. Attacks exploiting these communication stacks were demonstrated as Engine hours and VIN were changed.

The solution demonstrated in this work introduces a Secure Gateway to communicate directly with the PC diagnostics application using the existing vulnerable RP1210 system. Since all the messages flowing through the vulnerable system are encrypted using AES-128, previously effective cyberattacks are mitigated.

A key enabler for this approach to maintain the bandwidth and throughput of the existing J1939 network was the ability to operate in a point-to-point mode at 1 Mbps speed. This ensures that high busload operations, like reprogramming, maintain their current level of performance.

While the prototype introduced an additional hardware module in the Secure Gateway, alternative implementations are feasible if the logic can be implemented in existing modules. A key requirement for the system is to securely store the private keys needed for secure session key exchanges and digital signatures.

High quality and robust key management offline is challenging. A proposal for using encrypted PEM private keys and using the Secure Gateway as part of the decryption process was presented. This reduces the ability for an adversary to make use of any leaked key material.

The ATECC608A hardware security module is used primarily for key storage and secure key exchanges. The AES-128 algorithm is implemented in the processor. Should AES become outdated, a new algorithm could be implemented in its place. However, if the Elliptic Curve Cryptography becomes outdated, new hardware would be required. The ability to upgrade hardware-based cryptography and maintain crypto-agility is challenging when much of the structure of the messaging forces the implementation of 16-byte block ciphers and hardware-based security modules.

Overall, this approach promises a viable solution to address the concerns with cybersecurity vulnerabilities resulting from the implementation of RP1210-based diagnostics. This part-time connection to the vehicle is challenging to protect because maintenance operations are often trusting and the technician has physical access. By securing diagnostics communication, we greatly improve the cybersecurity posture of the heavy vehicle.

9. REFERENCES

[1] "RP1210: Windows API" in Recommended Practices Manual, American Trucking Association, Technology and Maintenance Council, 2016-2017.

[2] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, "Experimental Security Analysis of a Modern Automobile" IEEE Symposium on Security and Privacy, Oakland, CA, May 16–19, 2010.

[3] National Motor Freight Traffic Association, Inc, "A Survey of Heavy Vehicle Cyber Security," September 2015, Last accessed on June 1, 2020 from http://www.nmfta.org/documents/hvcs/nmfta heavy duty vehicle cyber security whitepaper v1.0.3.6.pdf

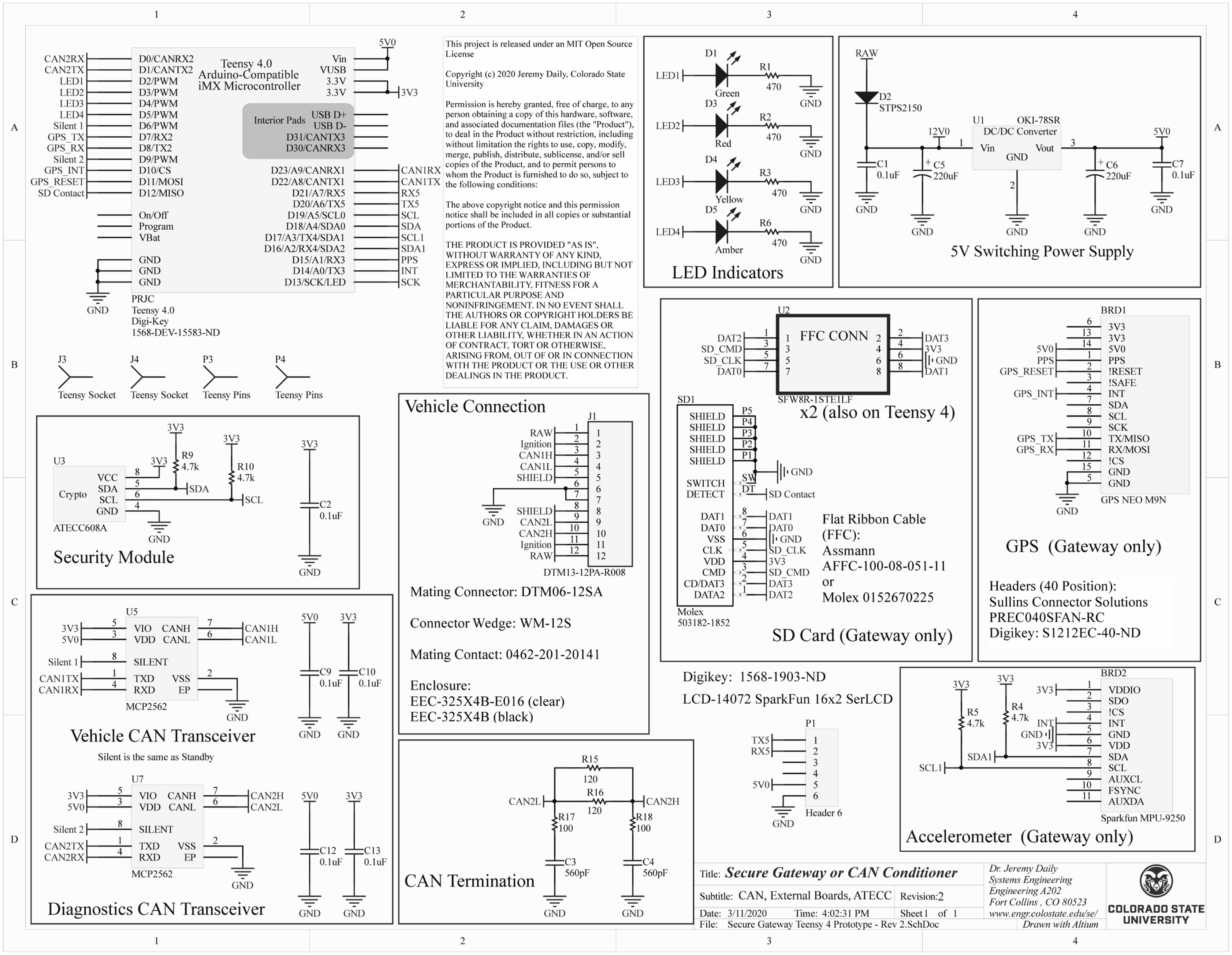
[4] Daily, J., Gamble, R., Moffitt, S., Raines, C. et al., "Towards a Cyber Assurance Testbed for Heavy Vehicle Electronic Controls," SAE Int. J. Commer. Veh. 9(2):339-349, 2016, https://doi.org/10.4271/2016-01-8142.

[5] "Automotive Gateway: A Key Component to Securing the Connected Car," NXP, Inc., Last Accessed 1 June 2020 from https://www.nxp.com/docs/en/white-paper/AUTOGWDEVWPUS.pdf

[6] Anderson, R. "Security Engineering" 3rd Ed., Wiley & Sons, 2020. Preview available at https://www.cl.cam.ac.uk/~rja14/book.html

10. APPENDIX

The appendix includes the schematic diagram of the Secure Gateway and some Python code snippets for establishing the secure session for the PC App.



This project is released under an MIT Open Source License

Copyright (c) 2020 Jeremy Daily, Colorado State University

Permission is hereby granted, free of charge, to any person obtaining a copy of this hardware, software, and associated documentation files (the "Product"), to deal in the Product without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Product, and to permit persons to whom the Product is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Product.

THE PRODUCT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE PRODUCT OR THE USE OR OTHER DEALINGS IN THE PRODUCT.

11. Partial Python Code Listing for the Secure Message Thread

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric import utils
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization

def setup(self):
    # Sets up the
    encrypted_key = None
    while encrypted_key is None:
        message_bytes = b'\x00' + struct.pack('>H', SESSION_ABORT)
        self.send_message(message_bytes)
        time.sleep(GATEWAY_REBOOT_TIME) #Wait for the gateway to reset
        (gateway_public_key,
             encrypted_key,
             init_vector,
             self.device_serial_number) = self.get_gateway_keys()
    private_key = self.get_private_pem_key()
    if private_key is not None:
        shared_secret = private_key.exchange(ec.ECDH(),gateway_public_key)
    else:
        return False

    backend = default_backend()
    key_cipher = Cipher(algorithms.AES(shared_secret[:16]),
                        modes.ECB(),
                        backend=backend)
    key_decryptor = key_cipher.decryptor()
    key = key_decryptor.update(encrypted_key) + key_decryptor.finalize()
    cipher = Cipher(algorithms.AES(key),
                    modes.CBC(init_vector),
                    backend=backend)
    self.decryptor = cipher.decryptor()
    self.encryptor = cipher.encryptor()
    return True
```

```
def get_gateway_keys(self, public_key_bytes=None):
    """We will send an ephemeral public key signed by a root of trust.
    The root of trust is a private key stored on the local computer that is
     envelope encrypted by a data key checked out from a server. The ATECC
     device on the gateway can verify the signature."""

    self.empty_rp1210()

    logger.debug("Requesting Device Serial Number.")
    message_bytes = b'\x00' + struct.pack('>H',REQUEST_SERIAL_NUMBER)
    self.send_message(message_bytes)
    device_serial_number = self.readNbytes(SEND_SERIAL_NUMBER,9)

    logger.debug("Requesting initialization vector.")
    message_bytes = b'\x00' + struct.pack('>H',REQUEST_INIT_VECTOR)
    self.send_message(message_bytes)
    init_vector = self.readNbytes(SEND_INIT_VECTOR,16)

    logger.debug("Requesting encrypted session key.")
    message_bytes = b'\x00' + struct.pack('>H',REQUEST_ENCRYPTED_KEY)
    self.send_message(message_bytes)
    encrypted_key = self.readNbytes(SEND_ENCRYPTED_KEY,16)

    logger.debug("Requesting device public key.")
    message_bytes = b'\x00' + struct.pack('>H',REQUEST_PUBLIC_KEY)
    self.send_message(message_bytes)
    gateway_public_key_bytes = self.readNbytes(SEND_PUBLIC_KEY_DATA,64)

    # Load the public key
    gateway_public_key = ec.EllipticCurvePublicNumbers(
                          int(gateway_public_key_bytes[:32].hex(),16),
                          int(gateway_public_key_bytes[32:].hex(),16),
                          ec.SECP256R1()).public_key(default_backend())
    logger.debug("Requesting public key signature.")
    message_bytes = b'\x00' + struct.pack('>H',REQUEST_SIGNATURE)
    self.send_message(message_bytes)
    gateway_signature = self.readNbytes(SEND_SIGNATURE,64)

    gateway_signature_der = utils.encode_dss_signature(
                            int(gateway_signature[:32].hex(),16),
                            int(gateway_signature[32:].hex(),16))
    try:
        gateway_public_key.verify(gateway_signature_der,
                            gateway_public_key_bytes,
                            ec.ECDSA(hashes.SHA256()))
        logger.debug("Good Signature")
        return (gateway_public_key, encrypted_key,
                init_vector, device_serial_number)
    except:
        logger.debug(traceback.format_exc())
        logger.debug("Bad Signature")
        return None, None, None
```