Interested in learning more about security?

SANS Institute InfoSec Reading Room

Hacking the CAN Bus: Basic Manipulation of a Modern Automobile Through CAN Bus Reverse Engineering

The modern automobile is an increasingly complex network of computer systems. Cars are no longer analog, mechanical contraptions. Today, even the most fundamental vehicular functions have become computerized. And at the core of this complexity is the Controller Area Network, or CAN bus. The CAN bus is a modern vehicle's central nervous system upon which the majority of intra-vehicular communication takes place. Unfortunately, the CAN bus is also inherently insecure. Designed more than 30 years ago, the CAN bus fai...

Hacking the CAN Bus: Basic Manipulation of a Modern Automobile Through CAN Bus Reverse Engineering

GIAC (GCIA) Gold Certification

Author: Roderick Currie, roderick.h.currie@gmail.com
Advisor: Stephen Northcutt
Accepted: May 18, 2017

Abstract

The modern automobile is an increasingly complex network of computer systems. Cars are no longer analog, mechanical contraptions. Today, even the most fundamental vehicular functions have become computerized. And at the core of this complexity is the Controller Area Network, or CAN bus. The CAN bus is a modern vehicle's central nervous system upon which the majority of intra-vehicular communication takes place. Unfortunately, the CAN bus is also inherently insecure. Designed more than 30 years ago, the CAN bus fails to implement even the most basic security principles. Prior scholarly research has demonstrated that an attacker can gain remote access to a vehicle's CAN bus with relative ease. This paper, therefore, seeks to examine how an attacker already inside a vehicle's network could manipulate the vehicle by reverse engineering CAN bus communications. By providing a reproducible methodology for CAN bus reverse engineering, this paper also serves as a basic guide for penetration testers and automotive security researchers. The techniques described in this paper can be used by security researchers to uncover vulnerabilities in existing automotive architectures, thereby encouraging automakers to produce more secure systems going forward.

1. Introduction

The Controller Area Network, or CAN bus, has been the core internal network bus for passenger automobiles for over 30 years. While networking technology has advanced significantly since CAN's introduction in the 1980s, the CAN bus itself has remained largely unchanged. According to automotive security researcher and author Craig Smith, "vehicle technologies haven't kept pace with today's more hostile security environment, leaving millions vulnerable to attack" (Smith, 2016). Since CAN predates the advent of the "World Wide Web" (CERN, 2013) and wireless networking protocols, it should come as no surprise that CAN was not designed to be secure from intrusion. What is surprising, however, is that automakers are still relying on such an archaic and inherently insecure platform in the era of on-board Wi-Fi, integrated cellular connectivity, Bluetooth, and even autonomous driving capability.

Now, more than ever before, as automakers push rapidly towards fully autonomous vehicles, it is critical that vehicle owners and passengers can trust that vehicles are secure from cyber-attack. High-profile vehicle security researcher Charlie Miller recently remarked that "in an autonomous vehicle […] the computers are now even more in charge" (as cited in Greenberg, 2017). Miller also noted that while today's cars allow the driver to override autonomous functions, the fully-autonomous car of the future will leave passengers "totally at the mercy of the vehicle" (as cited in Greenberg, 2017). Securing vehicular networks, therefore, *must* be given top priority by auto manufacturers. The stakes are simply too high to leave security as an afterthought.

There is widely agreed upon axiom in the information security industry that "security by obscurity is no security at all." And this certainly holds true when dealing with the CAN bus. The "security by obscurity" model has only lasted for this long because there is a general lack of published research exploring CAN bus vulnerabilities. The goal of this research project, therefore, is to demonstrate the overall insecurity of the CAN bus architecture and to provide a reproducible method for reverse engineering the CAN bus to encourage others to undertake similar research. By exposing vulnerabilities in existing automotive architectures, the security community can encourage automakers to produce more secure systems going forward.

roderick.h.currie@gmail.com

2. Recommended Reading

This paper covers the technical aspects of car hacking and CAN bus manipulation. For a more general, high-level overview of CAN bus technology, I recommend consulting my earlier GIAC Gold paper entitled *Developments in Car Hacking* (Currie, 2015) as a reference point and baseline. In addition to providing an overview of CAN, that paper explores some recent high-profile car hacking demonstrations and scholarly research on the topic of automotive hacking. I have also proposed some possible solutions to the security challenges facing CAN in my other recent GIAC Gold paper, *The Automotive Top 5: Applying the Critical Controls to the Modern Automobile* (Currie, 2016).

3. Why Hack Cars?

Before delving into the technical details of *how* to perform CAN bus hacking, it is important to first consider the rationale for hacking cars. Ethical hackers have targeted traditional computer systems for decades. These so-called "white hat" hackers perform attacks against systems in order to expose vulnerabilities so that systems may be better secured going forward.

The "security by obscurity" model, which is a favorite of the automotive industry, simply does not work. Rather than investing in proactive security solutions, automakers have a tendency to cut corners on security in favor of cost savings. This is a decision often made in the corporate boardroom. Until malicious vehicle hacking becomes more commonplace, automakers believe it does not warrant a significant amount of budget or attention. However, ignoring vulnerabilities or attempting to hide them does not lead to more secure systems. The only way to overcome this flawed model is through increased public exposure of security vulnerabilities. One of the goals of this paper is to increase awareness of the significant degree to which modern automotive systems are insecure.

Car hacking can also be thought of as a type of security audit. By auditing the security of one's own vehicle, it is possible to gain an improved understanding of the ways in which the vehicle might be vulnerable to attack and to take precautions accordingly. Most computer users would not trust a new web browser or a new operating

roderick.h.currie@gmail.com

system if they knew it had not undergone extensive penetration testing by the developer. Why, then, do we entrust our safety and the safety of our loved ones to automotive systems that are not audited for security by their manufacturers?

Attempting to "hack" a car can seem like a monumental task. But when done safely and with a solid understanding of the underlying systems involved, it can be a very rewarding experience. Performing a successful car hack is not only intrinsically rewarding, but when documented and shared with the security research community, also serves to promote the creation of more secure systems in the future. This paper provides the background knowledge needed to undertake a basic car hacking project centered around CAN bus manipulation. While the information and techniques covered in this paper may not be groundbreaking, it is my hope that this paper inspires others to undertake car hacking projects of their own to further the cause for vehicular cyber security. Historically, when vehicular security vulnerabilities have been disclosed by researchers, the auto manufacturers have then been forced to address the problems by better securing their vehicles. By uncovering vulnerabilities in vehicular systems, the security research community can have a positive overall impact on the field of automotive cyber security.

4. Existing Research

It must be emphasized that this project does *not* attempt to recreate a full vehicle attack model from start to finish. This project assumes that access to the vehicle's internal network has already been established. It is widely accepted in the automotive security research community that vehicles can be hacked through numerous different external interfaces, and that doing so is a relatively trivial undertaking.

In recent years, automotive cyber security has begun to receive increased public attention. This is due in large part to several high-profile examples of vehicle hacking that were picked up by the mainstream media. In 2011, a team of researchers from the University of Washington and the University of California, San Diego, successfully demonstrated that "remote exploitation is feasible via a broad range of attack vectors including mechanics' tools, CD players, Bluetooth and cellular radio" (Checkoway et al.,

roderick.h.currie@gmail.com

2011). This research brought to light just how many different potential points of entry were available on a typical, modern car.

A few years later, in 2015, security researchers Charlie Miller and Chris Valasek demonstrated a remote exploitation of an unaltered passenger vehicle via the vehicle's cellular interface. This attack took advantage of a vulnerability in the Sprint cellular network and the onboard Uconnect infotainment system of a 2014 Jeep Cherokee (Miller & Valasek, 2015). The attack allowed Miller and Valasek to remotely take over the Jeep's steering, transmission, and brakes, the aftermath of which is shown in *Figure 1*:

***Figure 1*: Jeep Cherokee in a Ditch after Brakes Were Disabled (Greenberg, 2015)**

In September 2016, a group of researchers from the Keen Security Lab successfully demonstrated an attack on a Tesla Model S. The team performed a wireless attack that required no physical access to the Tesla vehicle, and that ultimately allowed them to partially take over control of the vehicle. The attack required the Tesla to be connected to a malicious Wi-Fi hotspot, and took advantage of a vulnerability in the vehicle's integrated web browser (Golson, 2016). As modern vehicles such as the Tesla Model S incorporate connectivity features such as on-board Wi-Fi and integrated web browsers, this only serves to broaden the attack surface and create new potential points of entry for attackers.

roderick.h.currie@gmail.com

This project, therefore, seeks to build upon existing vehicle security research by exploring vulnerabilities *within* a vehicle's internal network rather than attempting to infiltrate the vehicle's perimeter defenses.

5. Hypothesis

The purpose of this project is to demonstrate that the modern internal vehicle network is generally insecure and can be manipulated with relative ease by utilizing a laptop computer, some inexpensive cables and adapters, and freely available software. Due to a lack of proper device authentication on the CAN bus, an ordinary laptop computer will be allowed to join and communicate on the CAN bus as if it were an authorized CAN controller. And due to a lack of message encryption, it will be possible to sniff and decode CAN messages to determine their function. Ultimately, sending modified CAN messages out over the CAN bus will result in the vehicle processing these reverse-engineered messages as if they were legitimate CAN traffic.

6. Legality of Car Hacking

Despite the various published reports of car hacking by security researchers, manipulation of automotive security systems is not without its legal ramifications. Before making any attempt to reverse engineer the CAN bus, it is important to understand the legality of doing so.

6.1. Volkswagen AG vs. the Security Community

In 2012, a group of security researchers from Radboud University in the Netherlands and the University of Birmingham in the United Kingdom discovered a significant security flaw in the engine immobilizer systems of vehicles from a handful of different manufacturers (Gallagher, 2015). In the spirit of information sharing, the team reached out to the automakers to inform them of the vulnerability. The team also shared their intent to publish their findings publicly at an upcoming security conference. This was worrying to the automakers, particularly Volkswagen AG. The list of impacted cars included vehicles from Volkswagen's Porsche, Audi, Bentley, and Lamborghini brands (Gallagher, 2015). Volkswagen was concerned that public disclosure of the vulnerability

roderick.h.currie@gmail.com

would benefit car thieves, and would require an expensive recall of all affected models to remediate the problem. Before the research team could publish its findings, however, Volkswagen filed a lawsuit to block the publication of the paper.

What followed was more than two years of litigation as Volkswagen used the legal system to keep the vulnerability undisclosed. In 2013, a British high court imposed an injunction on the researchers to legally prohibit them from sharing their findings (O'Carroll, 2013). The University of Birmingham, at the time, responded to the injunction by saying they were "disappointed with the judgment which did not uphold the defense of academic freedom and public interest" (O'Carroll, 2013). Similarly, Radboud University responded that "the decision of the English judge imposes severe restrictions on the freedom of academic research in a field that is highly relevant to society (cyber security)" (as cited in O'Carroll, 2013). Despite the team's research being of public importance, Volkswagen was unwilling to back down and face the financial burden associated with improving the security of its vehicles.

Bound by the judge's ruling, the research team was forced to withhold its findings and come to an understanding with Volkswagen before the injunction could be lifted. It was not until two years later, in 2015, that Volkswagen agreed to allow the team to publish a redacted version of their paper with certain, specific details of the attack removed. The paper, entitled "Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer" (Verdult, Garcia, & Ege, 2015), was presented at the USENIX security conference in Washington, DC that year.

6.2. The Digital Millennium Copyright Act (DMCA)

Until recently, car hacking – even for the purposes of security research – was illegal in the United States. The Digital Millennium Copyright Act (DMCA), which was signed by President Clinton in 1998, generally prohibits modifying copyrighted software or bypassing access control technologies (U.S. Copyright Office, 1998). While the DMCA was originally intended to protect publishers of traditional computer applications, it nonetheless also legally extended to the systems found in modern automobiles. Section 1201 of the DMCA effectively prohibits the reverse engineering of computer software for security research purposes, even if the researcher has purchased the software and owns

the device on which the software runs (Greenberg, 2016). In fact, John Deere, the well-known manufacturer of agricultural equipment, recently made a claim under the DMCA that farmers do not truly own their tractors, but rather receive “an implied license for the life of the vehicle to operate the vehicle” (as cited in Wiens, 2015). Similarly, General Motors commented to the U.S. Copyright Office that vehicle owners mistakenly “conflate ownership of a vehicle with ownership of the underlying computer software in a vehicle” (as cited in Wiens, 2015). The Digital Millennium Copyright Act has, therefore, long served as a deterrent to automotive security researchers.

Thankfully, in October of 2015, the U.S. Copyright Office signed into law a new series of exemptions to the DMCA that allow “good-faith” security research “in a controlled environment designed to avoid any harm to individuals or to the public” (as cited in Greenberg, 2016). Due to a one-year delay in implementation, the DMCA exemptions did not legally take effect until October 2016. Now that “car hacking” for the purposes of security research is no longer prosecutable under copyright law, there is no longer a looming fear of lawsuits hanging over the security research community. It is important to note that the DMCA exemptions still do not permit modification of a vehicle’s telematics or infotainment systems, nor do they permit modifications that would violate any other laws such as emissions regulations (O’Kane, 2015).

Predictably, auto manufacturers were opposed to the DMCA exemptions. Various organizations voiced objections to the Copyright Office, including the Association of Equipment Manufacturers, the Association of Global Automakers, the Auto Alliance, General Motors, and John Deere (O’Kane, 2015). The automakers claimed that unrestricted access to vehicles’ software could present "serious public health, safety and environmental concerns" (as cited in O’Kane, 2015). Security researchers have long recognized, however, that legislation such as the DMCA does not deter criminals or those with illegal or malicious intent. The DMCA, until recently, only served to stymie “good-faith” security research that would have benefitted consumers and improved the security of the auto industry as a whole.

roderick.h.currie@gmail.com

7. Safety First

Car hacking is inherently dangerous. When interacting with a vehicle's CAN bus, it is important never to lose sight of the fact that the target system is a two-ton metal object capable of reaching dangerous speeds in a short amount of time. Unlike traditional computer "hacking" where a mistake could lead to corruption of the operating system or a *Blue Screen of Death*, a car hacking mistake could lead to serious injury or *actual* death. Therefore, it is important to practice car hacking in a safe and controlled manner.

Because the CAN bus is where many of a vehicle's critical control units can be found, there is a very real possibility of provoking an unintended response from the engine, brakes, transmission, or other components of the vehicle while experimenting with CAN messages. Even if the engine or transmission are not the intended target systems, it is important to plan for the worst.

The best approach to safe car hacking is to raise the vehicle so that its driving wheels are no longer in contact with the ground, as seen in *Figure 2*. This will all but eliminate any concerns of unintended acceleration. This can be accomplished either by utilizing a vehicle lift, or by manually jacking up the vehicle and placing it on jack stands. It is important to become familiar with the target vehicle to know whether the vehicle sends power to the front wheels, rear wheels, or to all four.

***Figure 2*: A Car Safely Raised Off the Ground (Twelfth Round Auto, 2017)**

roderick.h.currie@gmail.com

It is also important to research – in advance – how the target vehicle can be shut off in the event of an emergency. Even with the car on jack stands, it is still important to have a shutdown plan to avoid damage from unintentionally over-revving the engine. For vehicles with older turn-key ignitions, turning the ignition to the "LOCK" or "ACC" position will shut off the car's engine regardless of what gear the car is in. But for newer cars with push-button ignitions, the "STOP/START" button will generally not respond to a single press if the vehicle is not in neutral or park. On these vehicles, the emergency shutoff procedure can vary from two or three quick presses of the ignition button, to a 3-second long hold of the button. Different makes and models of vehicles behave in different ways, but the owner's manual is typically a good place to find information on vehicle-specific emergency shutoff procedures.

8. Familiarization with the Target Vehicle

Just as it is essential to become familiar with the target vehicle for safety reasons, it is also wise to learn about the target vehicle's underlying systems before attempting to hack them.

As will be discussed later in this paper, the Controller Area Network (CAN) bus is a government-mandated standard found on almost all newer vehicles. However, to suggest that CAN bus hacking would yield the same results on all vehicles would be an oversimplification. The reality is that, beyond its mandatory diagnostic uses, CAN is implemented differently by each vehicle manufacturer. Some vehicles may utilize only one CAN bus, whereas others may have several, separate CAN buses. Many vehicles also feature other bus types, including LIN (Local Interconnect Network), FlexRay, MOST (Media Oriented Systems Transport), K-Line, SAE J1850, and more (Talbot & Ren, 2008). In fact, stumbling across manufacturer-specific proprietary bus types is not uncommon. It is, therefore, necessary to perform some background research on the target vehicle to learn what bus types are present.

Technical information such as bus types and bus locations can usually be found in third-party service manuals or automotive repair software resources such as ALLDATA

roderick.h.currie@gmail.com

(2017) or Mitchell1 (2017). While somewhat expensive, the data available from these resources can be extremely valuable to a vehicle security researcher.

In the interest of simplicity and scope control, this paper and the car hacking project it describes will focus solely on the CAN bus. This paper describes a technique for utilizing CAN tools to record, replay, and reverse engineer CAN messages to manipulate a vehicle. The CAN message reverse engineering techniques described in this paper go beyond existing research on the topic. The same general concepts presented here can be applied to other bus types, but additional tools and interfaces may be required.

9. About CAN

For a deeper exploration of the history and background of the Controller Area Network, please consult my previous GIAC Gold paper entitled *Developments in Car Hacking* (Currie, 2015). Nonetheless, this paper would be incomplete without at least a brief refresher of the CAN frame breakdown. *Figure 3* below shows the structure of a standard CAN data frame:

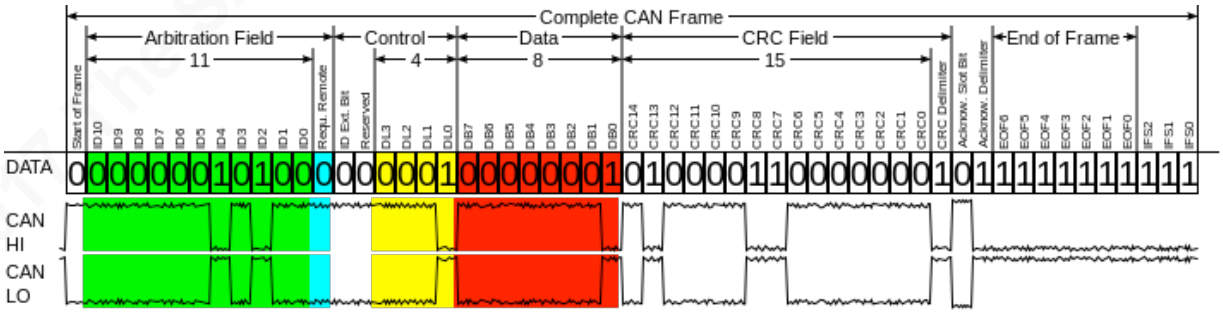

***Figure 3*: Complete CAN Frame (Wikipedia, 2014)**

For the purpose of the security testing being demonstrated in this paper, the main focus will be on the CAN arbitration ID (shown in green) and the CAN data (shown in red). The CAN arbitration ID is an 11-bit field that is used to identify different devices on the CAN bus and to prioritize messages. In order to craft CAN packets that the vehicle will process, it is first necessary to sniff out the valid CAN identifiers of devices on the bus. Later, those CAN identifiers can be reused to spoof legitimate devices. The other relevant field, the CAN data field, can be anywhere from 0 to 64 bits (8 bytes) in length. It is the CAN data field that tells the receiving device what function to perform. Gaining

roderick.h.currie@gmail.com

an understanding of the target vehicle's message format again requires sniffing legitimate traffic to decipher the vehicle's CAN message data.

10. Connecting via OBD-II

The Onboard Diagnostic port, also known as the OBD-II port, represents the most direct interface to a vehicle's CAN bus. Hacking a car via the OBD-II port has received criticism in the past because it is an unrealistic attack model. Certainly, there is a consensus among automotive security researchers that a real-world attacker is unlikely to have physical OBD-II access. However, there is also a consensus that the modern, connected car has a very broad attack surface and features a wide array of possible entry points. The remote exploitation of a vehicle is certainly more dramatic and tends to garner media attention. Nonetheless, performing research via direct, wired access to the OBD-II port is still a viable means of determining how a vehicle can be manipulated *after* access has been established, and should not be labeled as an unrealistic or incomplete model. Any manipulation of the CAN bus that can be performed via OBD-II can also be performed remotely against vehicles with remote CAN connectivity.

10.1. OBD-II Background

The California Air Resources Board (CARB) has long led the push for a self-diagnostic system to be required on all motor vehicles to aid in emissions testing and monitoring. In 1991, CARB rolled out an Onboard Diagnostic (OBD) mandate for all new vehicles sold at the time. However, there was no standard for the data link port, protocol, or port location. By 1994, CARB standardized the current iteration of OBD, known as OBD-II, and mandated that it be included on all new cars sold in California. By 1996, OBD-II was mandated nationwide in the United States (Lyons, 2015).

Included in the OBD-II specification is a special type of data link connector with a standard pinout. The standard OBD-II pinout includes a direct link to the CAN bus. For an automotive security researcher, the OBD-II port is essentially an unprotected backdoor into a vehicle's most sensitive embedded systems. While most backdoors are typically at least protected by a password, the OBD-II port is wide open to anyone with the appropriate hardware.

roderick.h.currie@gmail.com

On most vehicles, the OBD-II port is usually found underneath the dashboard. The OBD-II standard requires the port to be located within three feet of the driver and to be accessible without the need for tools (B&B Electronics, 2011). *Figure 4* shows where the OBD-II port can usually be found, and what it looks like:

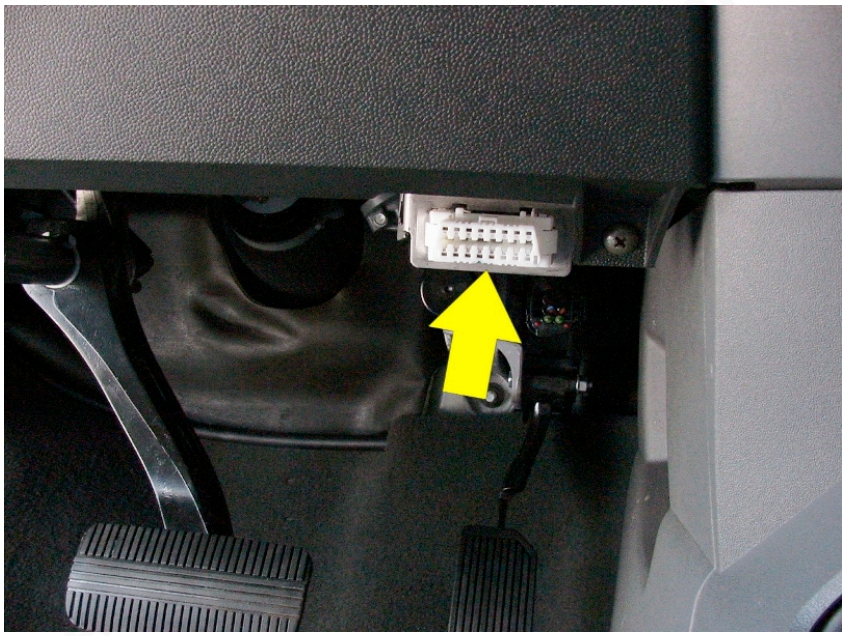

***Figure 4*: The OBD-II Port on a 2005 Nissan Titan (Nissanhelp, 2011)**

10.2. OBD-II Pinout

To better understand how the OBD-II port grants unrestricted access to the CAN bus, it is worth delving deeper into the OBD-II standard connector pinout. It should also be noted that CAN became a mandatory part of the OBD-II standard on all 2008 and newer light vehicles. Some older vehicles may not utilize CAN, but may have other, similar bus types instead. *Figure 5* shows a color-coded diagram of the OBD-II standard pinout and *Figure 6* lists the standard assignments for each pin:

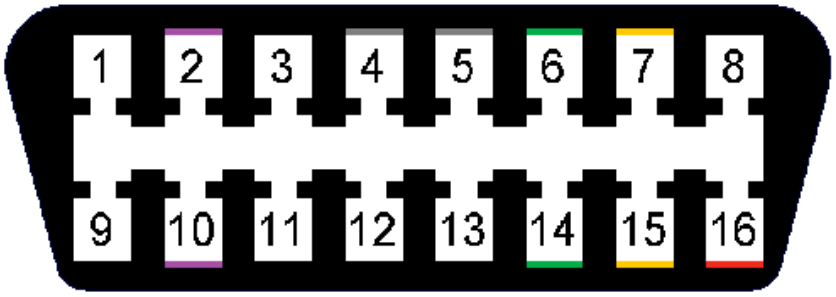

***Figure 5*: The OBD-II Pinout**

roderick.h.currie@gmail.com

Pin	Function	Pin	Function
1	Manufacturer Specific	9	Manufacturer Specific
2	**J1850 Bus (+)**	**10**	**J1850 Bus (-)**
3	Manufacturer Specific	11	Manufacturer Specific
4	Ground	12	Manufacturer Specific
5	Ground	13	Manufacturer Specific
6	**CAN High**	**14**	**CAN Low**
7	**K-Line (ISO 9141-2)**	**15**	**L-Line (ISO 9141-2)**
8	Manufacturer Specific	**16**	**12V Battery Power**

***Figure 6*: OBD-II Pin Assignments**

The OBD-II pinout diagram and table above highlight some of the important standard pins on the OBD-II connector. For the purposes of this paper, the J1850 bus (pins 2 and 10) and ISO 9141-2 bus (pins 7 and 15) are out of scope, but they are essentially diagnostic buses that operate in a similar manner to CAN. As noted in the table above, there are also various pins dedicated to manufacturer-specific bus types. Depending on the make and model of the vehicle being targeted, it may be necessary to delve into those other bus types to gain greater access to the vehicle's internal systems. Other noteworthy pins are pins 4 and 5, which are dedicated ground pins, and pin 16, which provides a constant supply of 12-volt power from the vehicle's battery. The purpose of pin 16 is to provide power to scan tools being plugged into the OBD-II port, so that they do not require an external power source.

The most important pins within the scope of this paper, however, are pins 6 and 14, which are dedicated to the Controller Area Network, or CAN bus. The terms "CAN High" (CAN_H) and "CAN Low" (CAN_L) are derived from the way in which CAN messages are physically transmitted. When the CAN bus is idle, both wires carry 2.5V of electricity. But when transmitting data, the CAN High wire increases to 3.75V and the CAN Low wire drops to 1.25V, creating a 2.5V voltage differential between the two wires (Nakade et al., 2015). It is this voltage differential on which CAN communication is based, and which makes the CAN bus so tolerant to electrical noise and interference.

roderick.h.currie@gmail.com

11. Car Hacking Hardware

Although the OBD-II port is an unsecured interface, car hacking via OBD-II is not quite as simple as plugging a computer directly into the OBD-II port. There *are* OBD-II scanners, commonly used by mechanics, which can be plugged directly into the OBD-II port. However, the capability of these devices usually does not extend beyond reading and clearing diagnostic codes.

In order to utilize a laptop computer to communicate on the CAN bus, some additional hardware peripherals are necessary. *Figure 7* below shows the hardware that was selected to communicate with the target vehicle:

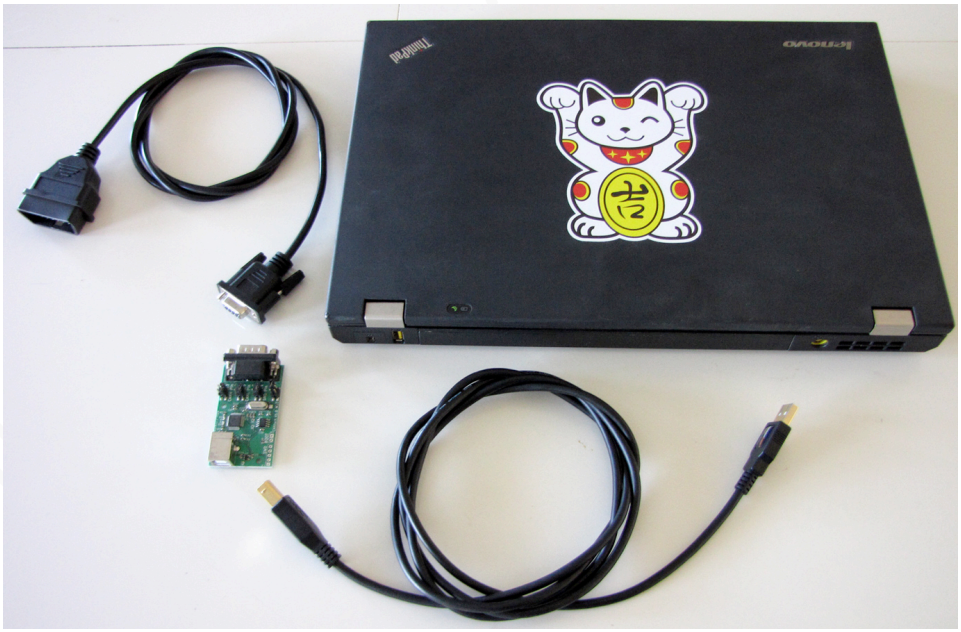

***Figure 7*: The Basic Hardware Required to Hack the CAN Bus**

11.1. Human Interface

Since car hacking via OBD-II requires a physical connection to the car, it is best carried out from inside the vehicle. Therefore, a laptop computer is necessary for sniffing and crafting packets. When selecting appropriate hardware, it is important to consider software compatibility. As will be discussed later in the paper, the chosen operating system for this car hacking project is Ubuntu 12.04. And so, it is critical to choose a laptop that can run this O/S in a stable manner without any hardware conflicts.

roderick.h.currie@gmail.com

The specific laptop ultimately selected was an older (circa 2011) Lenovo ThinkPad T420 (Lenovo, 2015). This laptop has an Intel Core i5-2520M 2.5GHz dual-core processor, 4GB of RAM, and a 128GB solid-state drive. Most importantly, the ThinkPad T420 has been certified by Ubuntu as supporting the Ubuntu 12.04 LTS 32-bit operating system (Canonical, 2011).

11.2. CAN to USB Interface

There are numerous CAN to USB interface products available on the market. Some are considerably more expensive than others. Because the device will act as the laptop computer's interface to the CAN bus, it is more complex than a simple adapter cable; it is an intelligent device that performs on-board processing of CAN packets. The interface of choice for this undertaking is CANtact, created by vehicle security researcher Eric Evenchick, and priced at $59.95 (Evenchick, 2015). CANtact was chosen for its affordability, cross-platform compatibility, and its open source nature. Throughout its development, Evenchick has kept the CANtact project entirely open source by making all of the design files, schematics, and code freely available online. *Figure 8* shows a close-up view of the CANtact CAN to USB interface device:

***Figure 8*: The CANtact CAN to USB Interface Device (Evenchick, 2015)**

CANtact can best be thought of as a network interface card, allowing the laptop computer to join and participate on the Controller Area Network. The CANtact device is seen by the computer's operating system as a CAN interface, and the computer is able to configure the settings of the interface as needed to communicate with the target vehicle.

roderick.h.currie@gmail.com

Once a link is established, the CANtact interface will pass CAN packets between the vehicle and the computer. CANtact draws its power from the computer via USB, so no external power source is required for the device.

11.3. Additional Cables

Getting the vehicle's CAN messages from the OBD-II port to the CANtact device requires an OBD-II to DB9 serial cable (SparkFun, 2015). This is nothing more than a "dumb" cable that swaps the OBD-II output to a serial pinout that can be received by the CANtact hardware device (or other hardware interface). The SparkFun OBD-II to DB9 cable puts the CAN High signal (OBD-II pin 6) onto DB9 pin 3, and the CAN Low signal (OBD-II pin 14) onto DB9 pin 5.

Once the CAN messages from the vehicle have been processed by the CANtact device, they are output through a USB Type B port. This, therefore, requires a USB-B to USB-A cable (Amazon, 2015) to pass the USB data from the CANtact interface to the laptop computer. USB-B to USB-A cables are also commonly used to communicate with printers, scanners, and other peripheral devices.

12. Software

With the appropriate hardware in place to bridge the gap between the laptop computer and the CAN-based systems of a modern automobile, it is also necessary to download and install the appropriate software to enable computer-to-vehicle communication.

12.1. Ubuntu 12.04

When selecting an operating system for a car hacking laptop, it is important to ensure that the O/S is fully compatible with the chosen computer system's hardware. Particularly when dealing with a Linux O/S, it should not be assumed that just any distribution of Linux can be installed on any given laptop. If a laptop was originally shipped with a Windows O/S, it is not uncommon to experience hardware incompatibility when redeploying Linux on the system. Ubuntu 12.04 LTS 32-bit was chosen due to its

roderick.h.currie@gmail.com

certification for the laptop being used in this project, the Lenovo ThinkPad T420 (Canonical, 2011).

It is also worth noting that car hacking does not explicitly require a Linux operating system. Car hacking can be performed on Windows. There are GUI-driven Windows-based applications available, such as CANdo from Netronics (2015) or CanKing from Kvaser (2014). However, in general, the vehicle security research community has favored Linux and, therefore, many of the open-source tools and devices available are designed primarily for use with Linux.

Another important reason to utilize Linux for car hacking is the Linux SocketCAN package. SocketCAN, which is explained in more detail below, is the Linux standard implementation of CAN protocols and drivers. SocketCAN has been part of the mainline Linux kernel since the release of Kernel 2.6.25 in April 2008 (Kernel.org, 2016). Therefore, in order to utilize SocketCAN, it is imperative to choose an O/S that runs Kernel 2.6.25 or later.

12.2. SocketCAN

Since 2008, the SocketCAN package has allowed Linux to offer native support for CAN devices at the network layer. SocketCAN, originally known as the "Low Level CAN Framework" (Hartkopp & Thürmann, 2006) was developed by Volkswagen AG and contributed to the Linux kernel as an open source framework. Among the stated goals of the Low Level CAN Framework project were the enablement of easy access between Linux applications and the CAN communication layers, and the creation of a package that was modular in design to enable reuse on other projects (Hartkopp & Thürmann, 2006). SocketCAN was also designed to make CAN communication "as far as possible similar to the ordinary use of TCP/IP sockets" (Hartkopp & Thürmann, 2006). And this is precisely what makes SocketCAN so versatile. By utilizing the tried and true Berkeley sockets API, which originated in 1983 (Kernel.org, 2016), CAN sockets in Linux behave in the same way as traditional TCP/IP sockets. This greatly reduces the learning curve when communicating with CAN devices. The diagram below in *Figure 9* shows how SocketCAN fits into the Linux networking stack:

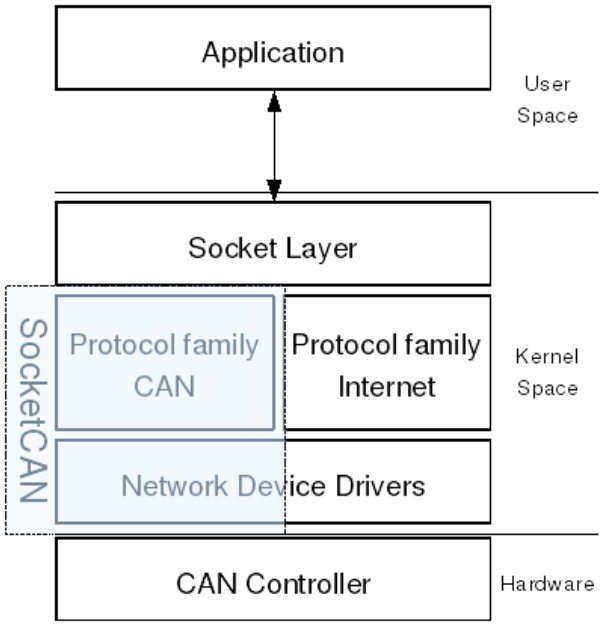

***Figure 9*: SocketCAN Implementation in Linux (Wikipedia, 2009)**

SocketCAN consists of two main parts: a protocol family, known as PF_CAN, and a collection of networking drivers for various CAN devices (Kleine-Budde, 2012). The PF_CAN protocol family is similar to the familiar PF_INET protocol family used for Internet protocol communication in Linux. SocketCAN also adds a new Ethernet protocol type, ETH_P_CAN, that allows CAN packets to be routed through the traditional Linux network layer (Hartkopp & Thürmann, 2006). This allows CAN network device drivers to implement the same standardized network driver model as Ethernet devices (Kleine-Budde, 2012).

In addition to the CAN device drivers found in the SocketCAN package, SocketCAN also provides a collection of useful user-space applications and utilities, known as *can-utils*, that can be very helpful for car hacking.

12.3. can-utils

The SocketCAN package includes an array of useful tools, but there are several can-utils that are particularly useful when communicating with the CAN bus of a modern vehicle.

12.3.1. candump

As its name implies, candump will dump all CAN packets directly to the console. When listing to an active vehicle's CAN bus, the raw output of candump will likely be overwhelming and of little use. However, candump accepts various filters to increase the usefulness of the output.

Usage: `candump can0`

12.3.2. cansend

Another utility with a relatively self-descriptive name, cansend allows a single packet to be sent out onto the CAN bus. When using cansend, it is necessary to specify the interface, the CAN identifier, and the CAN data.

Usage: `cansend can0 123#1122334455667788`

In the above usage example, a CAN frame will be sent with identifier 123 and data bytes 11, 22, 33, 44, 55, 66, 77, and 88 (Linklayer, 2016). The cansend utility assumes all values are hexadecimal.

12.3.3. cansniffer

Arguably the most useful of the can-utils when attempting to reverse engineer vehicle CAN bus messages, cansniffer works like candump but performs real-time filtering of the on-screen output. By filtering out any repetitive CAN messages containing data that remains unchanged, cansniffer displays only CAN messages for which the data is changing in real-time. This is particularly useful when performing CAN reconnaissance while physically operating the vehicle controls. For example, locking and unlocking the vehicle's doors while running cansniffer should make it easy to zero in on which messages and which bytes specifically control the locking functionality. By filtering out most of the "noise," cansniffer allows a security researcher to focus on only the relevant CAN packets.

Usage: `cansniffer can0`

roderick.h.currie@gmail.com

12.4. Wireshark with SocketCAN

Wireshark also offers support for SocketCAN devices. Wireshark can be used to display real-time CAN bus output, similar to candump, and can filter CAN messages based on message ID.

13. The Target Vehicle

This car hacking project will target the CAN bus of a 2011 Honda Civic LX 4-door sedan as pictured below in *Figure 10*:

***Figure 10*: 2011 Honda Civic LX Sedan**

While it is true that this car is already six years old at the time of writing this paper, the same techniques described herein can be applied to newer cars with similar results. Due to the federal OBD-II mandate, the vast majority of new vehicles feature a CAN bus architecture similar to that of the 2011 Honda Civic.

As mentioned earlier, knowing the target vehicle is essential. For this project, a paid subscription to an online repair data service was utilized to obtain wiring diagrams and schematics for the target vehicle. It was discovered that the 2011 Honda Civic sedan utilizes two separate CAN bus backbones: Fast CAN (F-CAN) and Body CAN (B-CAN). The F-CAN bus deals with the more critical components of the car, such as the engine, transmission, steering, brakes, and other fundamental vehicle control functions. The

roderick.h.currie@gmail.com

vehicle's gauge cluster can also be found on the F-CAN bus, as it relies on data being sent from some of the most critical components of the vehicle in order to provide accurate readouts to the driver. The 2011 Civic's F-CAN bus utilizes a pair of wires (CAN_H and CAN_L) and operates at 500Kbps. The B-CAN bus, on the other hand, utilizes only a single wire (SW-CAN) and operates at a much lower 33.33Kbps. The B-CAN bus handles the less critical functions such as the vehicle's radio, windows, door locks, comfort settings, and so on.

Because of the higher criticality and sensitivity of the components found on F-CAN, this car hacking project will focus on accessing the F-CAN bus to demonstrate just how vulnerable some of a modern vehicle's critical components are to unauthorized access and manipulation.

14. Limitations of CAN Hacking

When attempting to hack the CAN bus, there are numerous technical limitations that reduce the chances of success. This is particularly true of older vehicles where most vehicular functions are controlled via analog means. Ironically, it is the newer and more "advanced" cars that offer the most hacking potential, as they have greater interconnectivity between the various Electronic Control Units (ECUs) and a wider overall attack surface.

Even once access to the CAN bus has been established, spoofing CAN messages may not always yield the desired results. Charlie Miller and Chris Valasek explained this dilemma in their 2014 paper, *Adventures in Automotive Networks and Control Units* (Miller & Valasek, 2014). One of the vehicles Miller and Valasek targeted was a 2010 Ford Escape. The researchers found that although pressing the accelerator pedal created specific messages on the CAN bus, these messages when replayed over the bus did not result in acceleration of the vehicle. This is because many CAN messages are intended only to provide status information to other listening ECUs, but are not actually involved in the control of the vehicle. But as vehicles become increasingly interconnected, it is becoming more and more common that critical control functions such as steering, braking, and acceleration are accessible via the CAN bus.

roderick.h.currie@gmail.com

Miller and Valasek also highlighted another problem with hacking the CAN bus, which is that "there can be a lack of response or complete disregard for packets sent if there is contention on the bus" (Miller & Valasek, 2014, p. 29). The CAN bus is already an inherently busy network, but adding spoofed packets to the mix can often lead to unexpected results. It is important to remember when sending fake CAN messages that "the original ECU will still be sending packets on the network as well, [which] may confuse the recipient ECU with conflicting data" (Miller & Valasek, 2014, p. 29). Therefore, when performing CAN bus hacking, it is important to be prepared for a sometimes unpredictable response from the target vehicle.

15. Other Paths to the CAN Bus

Although many modern vehicles are designed without fundamental security principles in place, not all vehicles lack network segmentation on the CAN bus. Some vehicles place the OBD-II diagnostic module on a separate CAN bus to the more critical vehicle control modules. Due to the limitations of the OBD-II port on some vehicles, it may be necessary to find an alternative entry point to the CAN bus. However, this proved not to be a significant obstacle on the target vehicle. As with most vehicles, the 2011 Honda Civic has numerous locations where its F-CAN bus is readily accessible.

15.1. Accessing F-CAN

Some further examination of online repair diagrams revealed that one of the components on the Honda Civic's F-CAN bus is the Tire Pressure Monitoring System (TPMS) control module. Because the gauge cluster is on the F-CAN bus, and because the TPMS control module needs to send data to the gauge cluster (e.g. a low tire warning), it apparently made sense to the vehicle's designers to place the TPMS control module on the F-CAN.

Luckily for hackers and vehicle security researchers, the TPMS control module is conveniently located just below the steering column and can be easily disconnected. By disconnecting the TPMS control module, it is possible to hardwire into the TPMS module's CAN connector. *Figure 11* below shows what this process looks like:

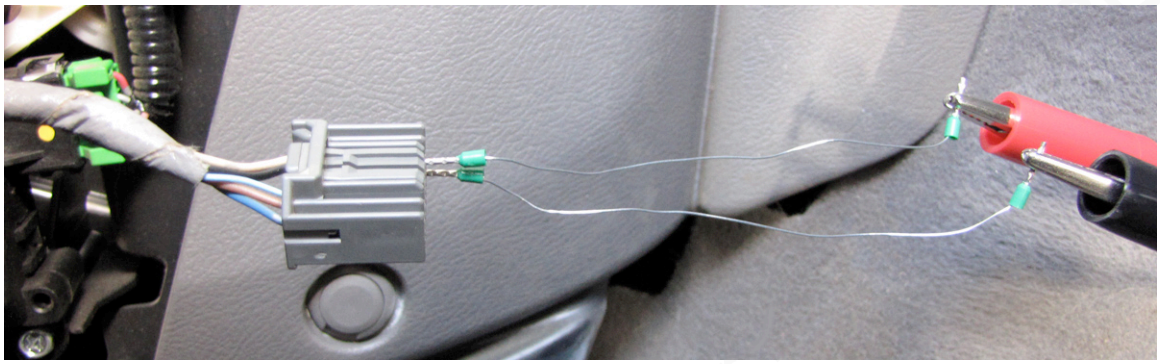

***Figure 11*: Hardwiring into the CAN Bus**

The main challenge with hardwiring into the TPMS module connector is knowing which PINS are used for the CAN signals. Through an examination of online repair data, it was determined that the TPMS module connector is a 20-pin connector with only 6 active pins. More importantly, the CAN-H signal is found on pin 2 and the CAN-L signal is found on pin 11. Then, temporary wire was used to access pins 2 and 11 on the TPMS connector, and alligator clips were used to connect back to the CAN-H and CAN-L pins (6 and 14 respectively) of the OBD-II to DB9 serial cable. From there, the rest of the peripherals and methods remain the same as when connecting directly to the OBD-II port.

15.2. Accessing B-CAN

Some additional research yielded a particularly attractive access point to the less critical B-CAN bus, shown below in *Figure 12*:

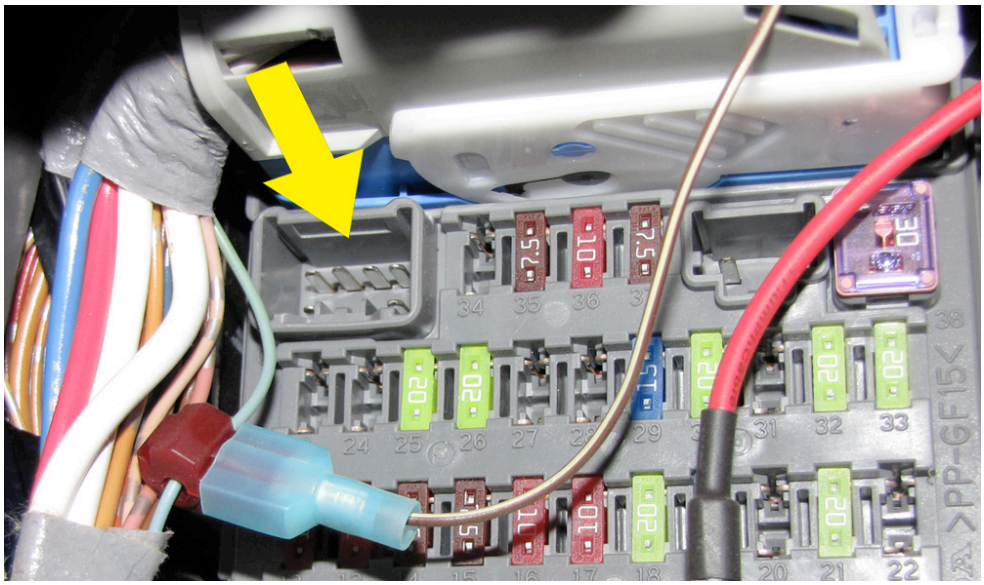

***Figure 12*: Option Connector Offering Physical Access to B-CAN**

roderick.h.currie@gmail.com

The highlighted area in *Figure 12* shows what the vehicle's repair literature describes as an "option connector" located in the fuse box under the dashboard. This so-called option connector is where an optional feature such as fog lights or navigation would be plugged in. Because the vehicle does not come with the option in question, the connector is unused. However, an examination of wiring diagrams reveals that pin 5 of the option connector carries the B-CAN signal. Because B-CAN runs on a single wire, an alligator clip to pin 5 is all that is required for direct access to the B-CAN bus.

16. CAN Bus Hacking Method

What follows is a reproducible method for configuring a standard laptop computer to communicate with the CAN bus of a modern vehicle. These steps are based on a clean installation of Ubuntu 12.04 LTS on a compatible laptop computer.

16.1. Install Dependencies

There are various utilities that must be installed before installing can-utils. The first of those is Git. Git is a distributed version control system used to manage software projects. Git, itself, has some dependencies that must first be downloaded and installed as shown below:

```
sudo apt-get install libcurl4-gnutls-dev libexpat1-dev
gettext libz-dev libssl-dev build-essential
```

Next, use wget to download the latest version of Git:

```
wget https://www.kernel.org/pub/software/scm/git/git-
2.12.2.tar.gz
```

Once downloaded, Git can be installed as follows:

```
tar -zxf git-2.12.2.tar.gz

cd git-2.12.2

make prefix=/usr/local all

sudo make prefix=/usr/local install
```

roderick.h.currie@gmail.com

Before attempting to install can-utils, there are some other build dependencies that must be installed:

```
sudo apt-get install autoconf automake pkg-config
libgtk-3-dev autogen libtool
```

16.2. Install can-utils

With all the necessary dependencies in place, the SocketCAN utilities, also known as can-utils, can be downloaded and installed:

```
git clone https://github.com/linux-can/can-utils.git
cd can-utils
./autogen.sh
./configure
make
sudo make install
```

16.3. Load Modules

Each time the operating system boots, it is necessary to load the required CAN modules to interface with the CANtact device:

```
sudo modprobe can
sudo modprobe can_raw
sudo modprobe slcan
```

It is also possible to set the modules to load automatically each time the O/S boots. This is advisable to save time later on:

```
sudo nano /etc/modules
```

When editing the modules file, each module (can, can_raw, and slcan) should be added to the file with each on its own line.

16.4. Set CANtact Jumpers

The CANtact interface device has a series of physical hardware jumpers that need to be set according to the type of connection being used. When utilizing an OBD-II to DB9 cable, the jumpers must be set to put CAN High on pin 3, CAN Low on pin 5, and the ground on pin 1. This jumper configuration is shown below in *Figure 13*:

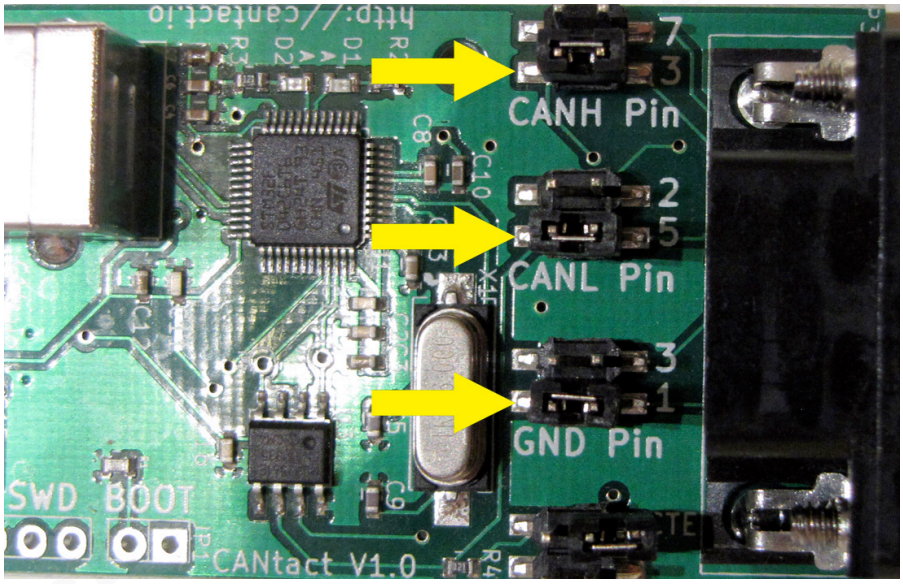

***Figure 13*: CANtact Jumper Placement for OBD-II to DB9**

16.5. Configure Interface

With all of the necessary software tools in place, it is at this point that the CANtact device can be physically connected to the laptop's USB port. When it is connected, CANtact will appear to the operating system as:

```
/dev/ttyACM0
```

The last digit may vary, depending on whether other USB devices are connected to the system. It is important to note the correct device name for the following step.

Binding the USB to CAN interface is accomplished with the following command:

```
slcand -o -s6 -t hw -S 3000000 /dev/ttyACM0 slcan0
```

The above command utilizes slcand, part of the SocketCAN package, which is a serial CAN device daemon that enables serial to CAN communication. The -o option is used to open the device for communication. The -t hw option specifies a hardware serial

flow. The -S option is used to specify the serial bitrate, in this case 3,000,000 bits per second or 3Mbps. This can generally be left unchanged. The name of the new interface in this case is slcan0, or serial CAN device zero. And, finally, the -s6 option is used to specify the vehicle's CAN bus bitrate. The -s6 option, specifically, signifies a CAN bus bitrate of 500Kbps. It is critically important that the correct CAN bus bitrate is selected for the target bus, otherwise the interface will be unable to communicate with the vehicle. The bitrate of the CAN bus will vary for different vehicles and different functional CAN buses. The CAN bus bitrate option should be chosen from the table below in *Figure 14*:

Option	Bitrate
-s0	10Kbps
-s1	20Kbps
-s2	50Kbps
-s3	100Kbps
-s4	125Kbps
-s5	250Kbps
-s6	500Kbps
-s7	800Kbps
-s8	1Mbps

***Figure 14*: Bitrate Options for slcand**

After binding the interface, it is also necessary to bring up the interface before it can be used. This is performed in the same manner as with an Ethernet interface:

```
ifconfig slcan0 up
```

With the link up, ifconfig should generate the following output as shown below in *Figure 15*. The new interface is now visible and its link state is reflected here.

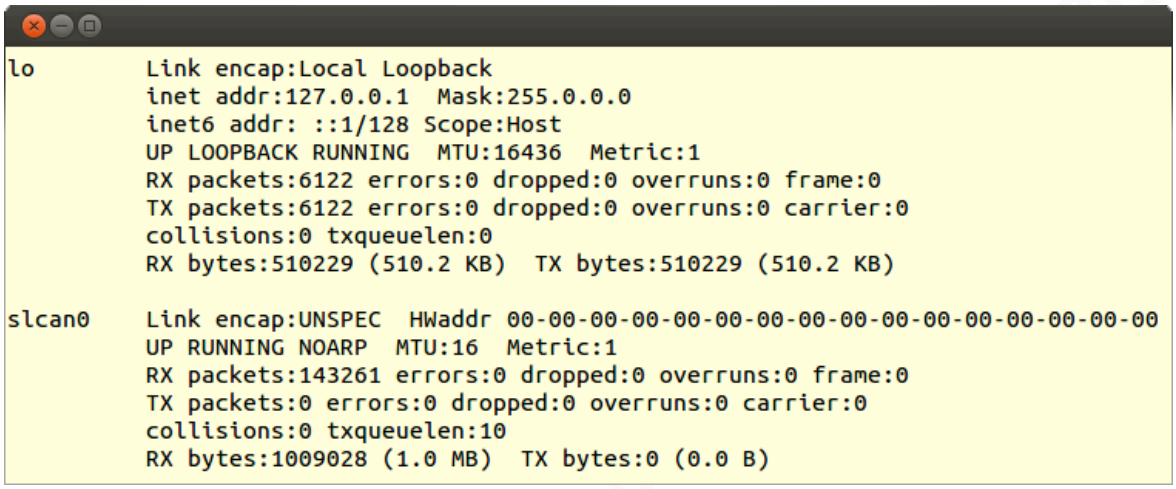

***Figure 15*: Output from ifconfig Showing slcan0 Interface**

The process of binding the interface and bringing up the link can also be automated to save time in the future (Walter, 2015). This is done by creating a udev rule to call a custom script:

```
sudo nano /etc/udev/rules.d/90-slcan.rules
```

The script should be populated with the following code:

```
ACTION=="add", ENV{ID_MODEL}=="CANtact_b20",
ENV{SUBSYSTEM}=="tty", RUN+=
"/usr/local/bin/slcan_add.sh $kernel"
```

The above code is specific to the ID of the CANtact device. If using a different type of adapter, the ENV{ID_MODEL} will need to be changed accordingly. This code calls a script named slcan_add.sh, which must first be created as follows:

```
sudo nano /usr/local/bin/slcan_add.sh
```

The slcan_add.sh file should be populated with the following content:

```
#!/bin/sh

slcand -o -s6 -t hw -S 3000000 /dev/$1 slcan0

sleep 2

ifconfig slcan0 up
```

Now, whenever the CAN serial device is connected to the computer, the operating system will automatically bind the interface at the correct serial bitrate and CAN bitrate, and will automatically bring the link up. Once a connection to the vehicle has been established, the reconnaissance phase can begin.

17. Performing CAN Reconnaissance

If all the prerequisite steps have been followed, the interface should be up and ready to communicate at the correct bitrate for the target vehicle's CAN bus. Now, the OBD-II end of the serial cable can be connected to the car's OBD-II port. To wake the CAN bus and start seeing CAN traffic, it may be necessary to turn the vehicle's ignition to the "ACC" or "ON" position, although it is not necessary to start the vehicle's engine.

A quick and easy test to see if CAN packets are being successfully received by the laptop computer is to run candump:

```
candump slcan0
```

At this point, a barrage of CAN messages should begin scrolling down the screen. Although the raw and unfiltered candump output is not very useful, it is nonetheless an effective way to confirm that the serial CAN interface is working correctly. An example of typical candump output is shown below in *Figure 16*:

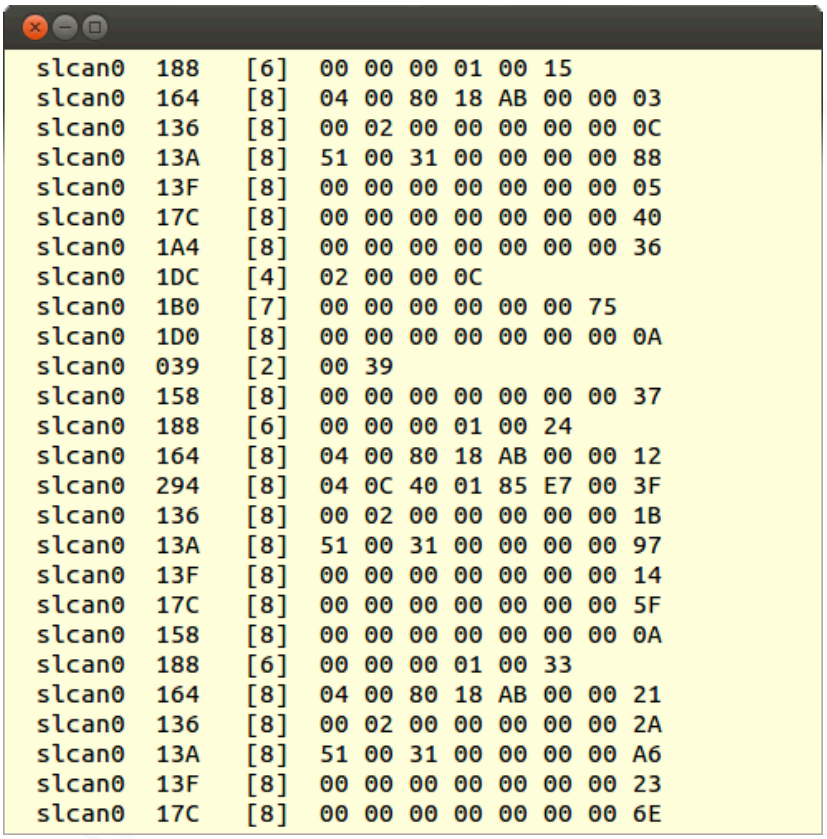

***Figure 16*: Unfiltered candump Output**

Clearly, sending raw candump output to the screen is not a practical way to observe the CAN bus. A far more useful way to make the CAN output manageable is to run cansniffer as follows:

```
cansniffer -c slcan0
```

As discussed earlier, cansniffer performs real-time filtering of the CAN messages it receives. Any messages that remain unchanged will be filtered out, thereby making the output more readable. Cansniffer displays only messages that are changing, and even goes one step further by highlighting the changing bytes in color when using the -c option. An example of the output from cansniffer is shown below in *Figure 17*:

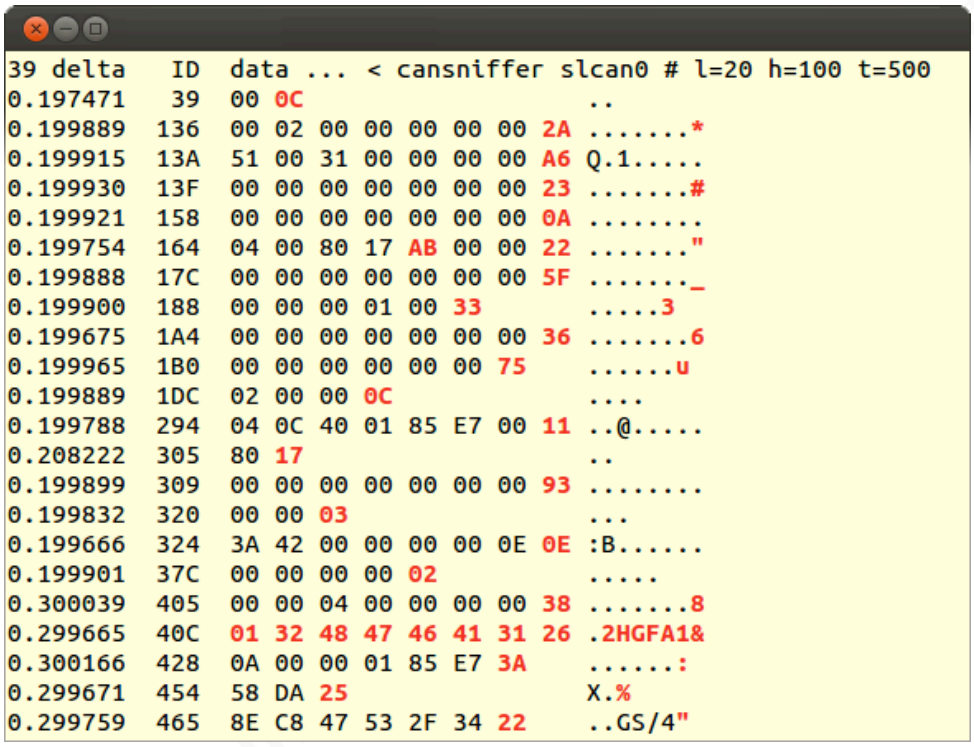

***Figure 17*: CAN Bus Output Filtered by cansniffer**

Another key advantage of cansniffer is that the output does not scroll. Each unique CAN ID is given its own line and remains fixed in that position. In the screenshot above, the specific bytes that are changing in real time are highlighted in red. By physically manipulating functions of the vehicle and observing the changing CAN bytes, it is possible to begin mapping out which message IDs and message data correspond to which specific vehicular functions. For example, shifting the gear lever from P (park) to R (reverse) causes a change in message ID 188. More specifically, byte 4 of message 188 changes from a value of 01 to a value of 02. Recording and mapping out these specific pieces of data is crucial to gaining a better overall understanding of how CAN communication takes place within the target vehicle.

17.1. Reconnaissance Findings

After a considerable amount of time spent experimenting on the 2011 Honda Civic and recording how the CAN messages changed, a table of vehicular functions and their corresponding CAN messages was built. Communications were successfully deciphered relating to the engine RPM, vehicle speed, gas pedal position, gear selection,

cruise control, headlights, turn signals, wipers, and more. A complete list of the CAN functions that were documented is provided in *Appendix A* at the end of this paper.

18. Replaying CAN Messages

With the CAN messages and their functions mapped out, the next logical step is to try replaying the messages back to the vehicle to see how the vehicle will respond. It is possible to send a single CAN message out onto the CAN bus by using the cansend command. The following example shows a message intended to display an engine RPM of approximately 8,000 on the tachometer:

```
cansend slcan0 1DC#023D1713
```

A more efficient method for sending CAN messages to the vehicle is to utilize canplayer, another of the utilities found in the SocketCAN package. The canplayer utility takes a candump log file (.log) and replays it back to the vehicle with the same original timing as the recording. To utilize canplayer, it is first necessary to generate a candump log file:

```
candump -l slcan0
```

When candump outputs to a log file, its output is different than the output displayed earlier when running candump without the -l option. The generated log file is formatted in a way that canplayer can read and interpret. Another very useful feature of candump is the ability to log only messages with a certain CAN ID. This feature is utilized heavily later in this paper. For example, logging only messages relating to vehicle RPM data (message ID 1DC) can be accomplished in the following way:

```
candump -l slcan0,1DC:7FF
```

The 7FF in the command above tells candump to also record any extended CAN frames (EFF) and any remote transmission request (RTR) messages. This way, every message with the desired CAN ID of 1DC will be recorded in full. The resulting .log file can then be replayed back to the vehicle with canplayer by utilizing the following syntax:

```
canplayer -I candump-2017-04-05-183520.log
```

roderick.h.currie@gmail.com

When using canplayer to play back a candump log file, canplayer will automatically use the same CAN interface from which the candump file was recorded. The interface is defined in the log file.

19. Customizing CAN Playback Files

Crafting customized CAN messages by hand is relatively easy. For example, the cansend utility allows custom CAN messages to be manually typed into the console and sent out over the CAN bus. However, sending individual messages in this manner is unlikely to allow for much control over the vehicle. This is because when a CAN message is received and processed, its effects usually only last for 10 to 20 milliseconds before another CAN message is received and processed by the listening controller. Therefore, when attempting to control certain vehicular functions, a continuous stream of well-timed CAN messages is required.

One way to continuously stream CAN messages to the vehicle is by utilizing a scripting language such as Python. With this method, a loop can be created to execute a cansend command at a specified interval, such as every 20ms. An alternative way to send a steady stream of CAN messages is to build a custom .log file to be played back to the vehicle using canplayer. The .log file must be in the following format:

```
(1492287144.880000)  slcan0  1DC#023D1713
(1492287144.900000)  slcan0  1DC#023D1713
(1492287144.920000)  slcan0  1DC#023D1713
(1492287144.940000)  slcan0  1DC#023D1713
(1492287144.960000)  slcan0  1DC#023D1713
(1492287144.980000)  slcan0  1DC#023D1713
(1492287145.000000)  slcan0  1DC#023D1713
```

***Figure 18*: Excerpt From a CAN .log File**

The first column of the .log file represents the current date and time in what is known as “epoch time” (EpochConverter, 2017). This is the number of seconds that have elapsed since midnight GMT on January 1st, 1970. When working with .log files in canplayer, the time itself is not important; what is important is the time increment from one line to the next. When playing back a .log file, canplayer plays the first line

immediately and then waits the amount of the incremental time difference before playing the next line. In the example shown above in *Figure 18*, each line is played at an interval of 0.02 seconds, or 20 milliseconds.

The second column of a CAN .log file represents the interface over which the CAN message should be sent. For the example in *Figure 18*, this is set to slcan0 and should not be modified. Finally, the third column of the .log file is the CAN message. The first three digits of the CAN message represent the message ID. The "#" symbol is necessary to separate the message ID from the message body. The characters that follow the "#" symbol represent the hexadecimal CAN message data.

The message in the example in *Figure 18* tells the tachometer to display a reading of 2,000 RPM. Because the message is repeated seven times at an interval of 20ms, its effect would be expected to last 140ms, or just a fraction of a second. Therefore, if a canplayer .log file is to have any significant impact on the target vehicle then it must usually be hundreds – or even thousands – of lines in length.

For the purposes of this research, one of the most efficient ways to create lengthy custom CAN .log files was to utilize Microsoft Excel and its built-in formula functionality. An example of the formula used to generate the timestamps in *Figure 18* is shown below in *Figure 19*:

	A
1	(1492287144.880000)
2	="("&TEXT(ABS(A1)+0.02,"0.000000")&")"
3	="("&TEXT(ABS(A2)+0.02,"0.000000")&")"
4	="("&TEXT(ABS(A3)+0.02,"0.000000")&")"
5	="("&TEXT(ABS(A4)+0.02,"0.000000")&")"
6	(1492287144.980000)
7	(1492287145.000000)
8	(1492287145.020000)

***Figure 19*: Excel Formula for CAN Timestamps**

In the above example, the increment of 0.02 seconds could easily be changed to 0.01 for 10ms, or 0.005 for 5ms, and so on. The correct formula only needs to be entered on a single line, then Excel's built-in click-and-drag functionality allows the formula to be easily extended across hundreds or thousands of consecutive rows. This method of

roderick.h.currie@gmail.com

.log file creation is less time-consuming than using a programming language to script the creation of CAN playback files.

Once the timestamps are worked out, all that remains is to copy and paste the interface and message data across all rows of the file. In Excel, the playback file should be saved as a tab-delimited text file (.txt) and then renamed with a .log extension for playback by the canplayer utility. Knowing the specific message data to use for an intended task can be a process of trial and error, but observing and documenting patterns in known good data is an advisable starting point.

20. Manipulation of the Target Vehicle

The ultimate goal of this project was not only to decipher and document CAN message functions, but also to exert some degree of control over the target vehicle. As it turned out, this could be accomplished with relative ease. What follows is a breakdown of the different ways in which the vehicle was successfully manipulated.

When sending CAN messages to the target vehicle, any physical access point on the appropriate CAN bus can be used. When communicating with this vehicle's F-CAN, the results were the same regardless of whether connecting via the OBD-II port or tapping into the vehicle's TPMS module connector. This only serves to highlight the lack of secure network segmentation found in today's vehicles.

In each case below, a reverse-engineered CAN log file (.log) was created and played back to the vehicle utilizing canplayer with the following syntax:

```
canplayer -v -I filename.log
```

When running canplayer, the -v option (verbose) displays each line of the .log file on the screen in real time, as it is being played back. The -I option is used to specify the filename of the input file.

In each case, the CAN .log file can either be played back to the vehicle while the engine is running or while the engine is off (as long as the ignition is in the ON position). Regardless of whether the vehicle is parked or is in motion, it is still possible to take control of the CAN bus.

roderick.h.currie@gmail.com

21. Manipulating Engine RPM Data

The target vehicle, a 2011 Honda Civic, features a traditional tachometer gauge with a needle to represent engine RPM. Listening to CAN traffic and deciphering message functions revealed that the engine RPM data being displayed on the tachometer comes from CAN messages with an ID of 1DC. Under normal conditions, the engine's Powertrain Control Module (PCM) broadcasts a 1DC message on the CAN bus every 20 milliseconds, or 50 times per second. The gauge cluster's control unit constantly listens on the CAN bus for messages with this identifier. Every time a message with an ID of 1DC is received, the needle position on the tachometer is updated accordingly.

The following is what a typical RPM CAN message looks like:

```
1DC#0212B824
```

On the target vehicle, and on many other Honda vehicles, the CAN message ID of 1DC is used exclusively to broadcast engine RPM data on the CAN bus. In this case, the 1DC message is 4 bytes in length. Throughout this research, it was found that the first byte never changed from a value of 02. The remaining three bytes, however, are a hexadecimal representation of the actual engine RPM after being encoded using a basic algorithm. After some experimentation, it was found that the following method depicted in *Figure 20* can be used to derive the approximate human-readable engine RPM from the hexadecimal data:

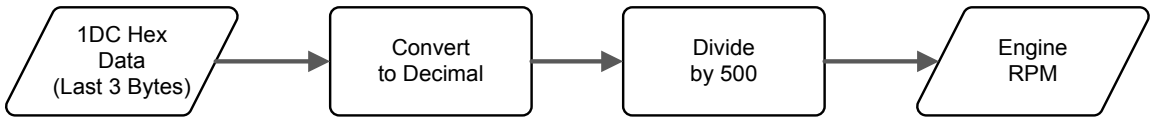

***Figure 20*: Converting 1DC CAN Data to Human-Readable Engine RPM**

For example, a 1DC message with the last three bytes of 12B824 converts to a decimal value of 1,226,788. This number divided by 500 yields a result of 2453.576, or approximately 2,450 RPM. In testing, when a CAN message of 1DC#12B824 was played back to the vehicle continuously, the tachometer needle would consistently move to a position of approximately 2,450 RPM, thereby validating the above algorithm.

Reverse engineering CAN RPM messages can be accomplished by picking a

roderick.h.currie@gmail.com

desired RPM and running it through the above algorithm in reverse. Interestingly, it was found that the vehicle would only process a CAN RPM message if the corresponding decimal value ended in either 03 or 53; all other messages were dropped without being processed. This is presumably a measure to promote fault-tolerance. The table shown below in *Figure 21* lists CAN messages for RPM values ranging from 1,000 through 8,000:

1DC Message	Last 3 Bytes	Decimal	Divide by 500	Approx. RPM
0207B1EF	07B1EF	504303	1008.606	1,000
020F3C03	0F3C03	998403	1996.806	2,000
021754DD	1754DD	1529053	3058.106	3,000
021EF503	1EF503	2028803	4057.606	4,000
022630F7	2630F7	2502903	5005.806	5,000
022E5331	2E5331	3035953	6071.906	6,000
0235DD45	35DD45	3530053	7060.106	7,000
023D1713	3D1713	4003603	8007.206	8,000

***Figure 21*: 1DC CAN Messages and Their Corresponding Engine RPM Values**

Using the values shown in the table above, a custom .log file was created which, when played back to the vehicle, yielded full control of the tachometer. The final RPM .log file created for this project contained 600 individual CAN messages and resulted in 12 seconds of uninterrupted control of the tachometer. It is possible to create even longer files or to use a scripting language such as Python to establish indefinite control of CAN bus controllers. However, a 12-second demonstration is sufficient to show successful manipulation of the vehicle.

A short excerpt from the CAN .log file is provided below in *Figure 22*, showing the overall technique of repeating 1DC RPM messages every 20 milliseconds and varying the data bytes to display different RPM values:

```
(1492287146.060000)  slcan0  1DC#023D1713
(1492287146.080000)  slcan0  1DC#023D1713
(1492287146.100000)  slcan0  1DC#023D1713
(1492287146.120000)  slcan0  1DC#023D1713
(1492287146.140000)  slcan0  1DC#023D1713
(1492287146.160000)  slcan0  1DC#023D1713
(1492287146.180000)  slcan0  1DC#023D1713
(1492287146.200000)  slcan0  1DC#023D1713
```

roderick.h.currie@gmail.com

```
(1492287146.220000)  slcan0     1DC#022E5331
(1492287146.240000)  slcan0     1DC#022E5331
(1492287146.260000)  slcan0     1DC#022E5331
(1492287146.280000)  slcan0     1DC#022E5331
(1492287146.300000)  slcan0     1DC#022E5331
(1492287146.320000)  slcan0     1DC#022E5331
(1492287146.340000)  slcan0     1DC#022E5331
(1492287146.360000)  slcan0     1DC#022E5331
(1492287146.380000)  slcan0     1DC#022E5331
(1492287146.400000)  slcan0     1DC#022E5331
```

***Figure 22*: Excerpt From Tachometer Attack .log File**

Although it is not possible to capture the moving tachometer needle in a static image, the photograph provided below in *Figure 23* shows the needle positioned at 8,000 RPM – something that would be impossible under normal conditions due to the vehicle's built-in rev-limiter:

***Figure 23*: Successful Attack on Tachometer**

Additionally, a video of the tachometer attack from this project is available online at the following URL: https://www.youtube.com/watch?v=euRyJCgfRGo

22. Manipulating Vehicle Speed Data

The technique used to manipulate the vehicle's speedometer is similar to that used to manipulate the tachometer. However, vehicle speed data uses a different CAN message

roderick.h.currie@gmail.com

ID and a more complex message structure. Through a process of elimination, it was found that the message ID for speedometer data for the target vehicle is 158. A normal 158 CAN message is shown below:

```
158#03BE03D803C7022C
```

Unlike the 1DC messages for RPM, which have a length of 4 bytes, a normal 158 CAN message has a length of 8 bytes. Also, whereas 1DC RPM message are broadcast every 20ms, 158 messages are broadcast every 10ms, or 100 times per second. After a considerable amount of road testing and experimentation, it was determined that a 158 CAN message can be divided into five distinct parts as follows:

Bytes 1 & 2: Speed data for purposes other than speedometer.
Bytes 3 & 4: Engine RPM data for purposes other than tachometer.
Bytes 5 & 6: Speed data for display on speedometer.
Byte 7: Signal to increment odometer.
Byte 8: Signal to indicate vehicle is in motion.

Much of the reverse engineering of CAN 158 messages was performed using Microsoft Excel and its built-in formula and graphing functions. An effective way to view and interpret a stream of 158 messages is to break them down into their different functional parts. Excel formulas allow the different segments of the message to be easily broken out and modified independently of each other. *Figure 24* below shows a small excerpt from a 158 candump .log file after it has been dissected in Excel:

	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
CAN Message	B1 & B2	B1 & B2	B3 & B4	B3 & B4	B5 & B6	B5 & B6	B7	B7	B8	B8
158#060C0633061A0236	060C	1548	0633	1587	061A	1562	02	2	36	54
158#06100636061D020E	0610	1552	0636	1590	061D	1565	02	2	0E	14
158#0614063906210211	0614	1556	0639	1593	0621	1569	02	2	11	17
158#061C063F062A0229	061C	1564	063F	1599	062A	1578	02	2	29	41
158#061F0644062D023C	061F	1567	0644	1604	062D	1581	02	2	3C	60
158#061E0647062C020E	061E	1566	0647	1607	062C	1580	02	2	0E	14
158#06220648062F0313	0622	1570	0648	1608	062F	1583	03	3	13	19
158#0628064E0636032E	0628	1576	064E	1614	0636	1590	03	3	2E	46
158#062C065206390331	062C	1580	0652	1618	0639	1593	03	3	31	49
158#062C0653063A0302	062C	1580	0653	1619	063A	1594	03	3	02	2

***Figure 24*: Breaking Down the 158 Message Structure**

roderick.h.currie@gmail.com

An even better way to fully understand and analyze just what the 158 CAN message does is to visually depict its various byte pairs using a line graph as shown below in *Figure 25*. This allows for pattern analysis, which is an essential method for learning how the target vehicle communicates.

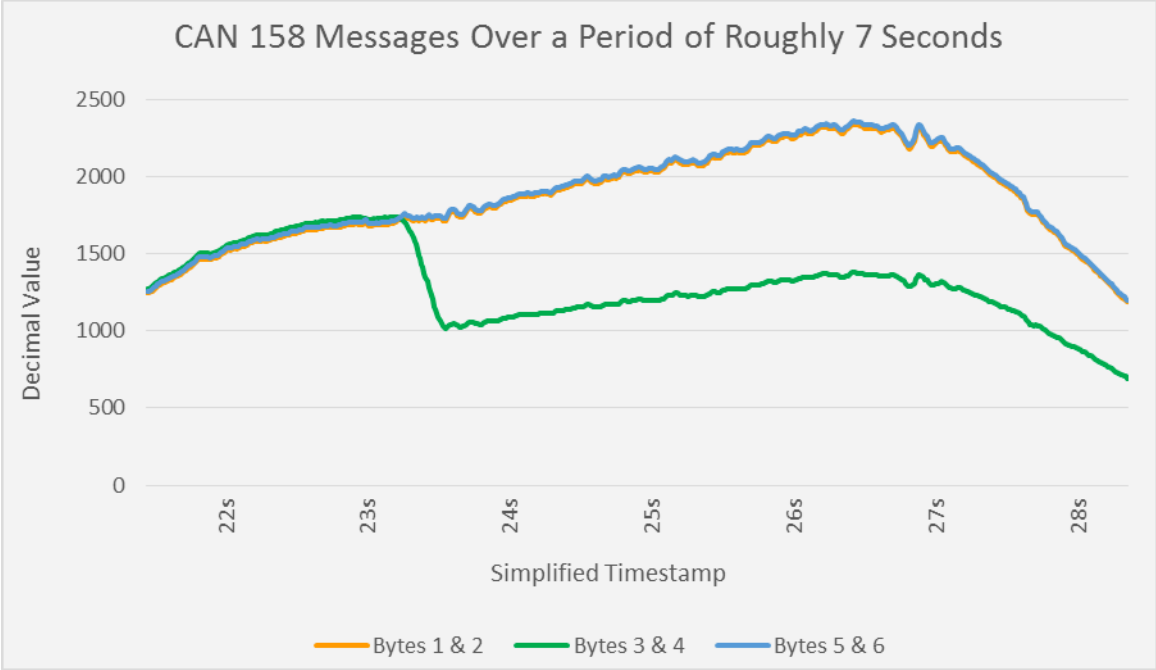

***Figure 25*: Charting the Different Parts of a CAN 158 Message**

The line graph above in *Figure 25* was created using an approximately 7-second excerpt of CAN messages with ID 158 recorded from a longer session of normal driving behavior. The "epoch" timestamps have been removed for simplicity and replaced with a timestamp showing the number of seconds that have elapsed since the recording began.

The graph immediately offers up evidence regarding what is happening within the different byte pair segments of a CAN 158 message. As indicated by the blue and yellow lines, byte pairs 1 & 2 and 5 & 6 both report the speed of the vehicle and mirror each other very closely. The green line of bytes 3 & 4 represents engine RPM data that is *not* used by the tachometer and is entirely separate from the 1DC RPM messages discussed earlier in the paper. As the graph above shows, the amount of work being performed by the engine was directly commensurate with the vehicle's speed until shortly after the 23-second mark. The green line's sharp drop-off represents a gear change; the engine RPM

decreased significantly, but then continued to climb again as the car accelerated. The engine RPM and vehicle speed lines then continued to roughly parallel each other throughout the experiment when driving on a level surface.

Bytes 7 of a 158 CAN message is used exclusively by the odometer and will be covered in greater detail later. Byte 8, however, has some relevance to vehicle speed data and is worth explaining at this time. The data being transmitted in byte 8 may initially seem quite perplexing. While speedometer data is clearly contained in bytes 5 & 6, it is not possible to manipulate the speedometer without byte 8 also being present. Similarly, although it was determined that byte 7 is used for the odometer signal, the odometer will not update unless byte 8 also contains data. Analysis of 158 CAN packets during normal driving reveals that the byte 8 data has no direct correlation to the vehicle's speed or engine RPM. The byte 8 data appears to fluctuate regardless of how the vehicle is being driven, so reaching a logical conclusion about the purpose of byte 8 was not easy.

It can be concluded that the true purpose of byte 8 is essentially to let the vehicle's ECUs know that the vehicle is in motion. Rather than containing direct numerical data, the messages contained in CAN 158 byte 8 are actually more of a "pulse" type signal. And this pulse signal varies depending on the state of the vehicle. This is best illustrated in the graphs below in *Figure 26*:

***Figure 26*: 158 Byte 8 "Pulse" Signal: Stationary (L) vs. Moving (R)**

The two graphs above represent two separate samplings of byte 8 data each created from 30 consecutive CAN 158 messages. The left set of data was recorded when the vehicle was parked, whereas the right data set was recorded while the vehicle was being driven.

roderick.h.currie@gmail.com

As can be seen from the left graph, the 158 byte 8 signal is uniform when the vehicle is not in motion. The signal repeatedly peaks at a decimal value of 61, before dropping down to a value of 0. This occurs consistently regardless of whether the engine is running or not, and regardless of whether the vehicle is in gear. When the vehicle is being driven, however, the byte 8 signal immediately becomes a lot more erratic. More crucially, the signal never reaches the high value of 61 or the low value of 0 associated with the stationary signal. It is this slightly narrower signal range that tells the vehicle it is in motion, thereby prompting the gauge cluster to process speedometer messages (bytes 5 & 6) and odometer messages (byte 7). When the byte 8 signal tells the car it is stationary or if the byte 8 signal simply is not present, then bytes 5, 6 and 7 are all ignored by the listening ECUs.

Focusing in on the speed data of bytes 5 & 6, after some experimentation, the algorithm shown below in *Figure 27* was established for deriving the vehicle speed in miles per hour (mph) from the original CAN hexadecimal values:

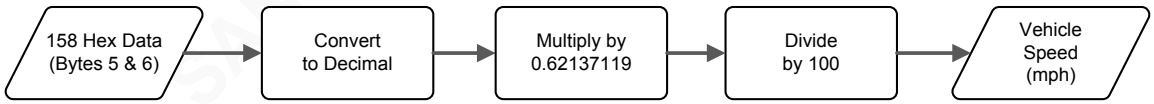

***Figure 27*: Converting 158 CAN Data to Vehicle Speed in Miles per Hour**

On the target vehicle, and on most vehicles, the ECUs process vehicle speed data in kilometers per hour (km/h). The fact that the gauge cluster displays speed in miles per hour (mph) is merely a display setting. Therefore, it is necessary to multiply the decimal km/h data by 0.62137119, as 1 kilometer is equal to 0.62137119 miles. It is also necessary to divide the resulting value by 100 to position the decimal point correctly.

For example, a hexadecimal speed value of 2659 can be converted to 9817 in decimal. When 9817 is multiplied by 0.62137119, the result is 6100.00097223. When this number is divided by 100, the result is 61.0000097223, or a speed of 61 mph. The table below in *Figure 28* lists the CAN 158 byte 5 & 6 data associated with various other vehicle speeds:

roderick.h.currie@gmail.com

158 Bytes 5 & 6	Decimal	Multiply by 0.62137119	Divide by 100	Approx. Speed
064A	1610	1000.40761590	10.00407616	10mph
12DD	4829	3000.60147651	30.00601477	30mph
1F6F	8047	5000.17396593	50.00173966	50mph
324B	12875	8000.15407125	80.00154071	80mph
4B71	19313	12000.54179247	120.00541792	120mph
6E04	28164	17500.29819516	175.00298195	175mph
7DBB	32187	20000.07449253	200.00074493	200mph

***Figure 28*: 158 CAN Messages and Their Corresponding Vehicle Speed Values**

The method used to demonstrate continuous manipulation of the vehicle's speedometer was similar to that used earlier for the tachometer. A custom .log file was created and canplayer was used to play it back to the vehicle, overriding the vehicle's true speed data and replacing it with spoofed data. This attack proved possible regardless of whether the vehicle was in motion or not. The same attack could also be accomplished programmatically, by running a custom script against the vehicle rather than using a saved CAN .log file. It is not practical or realistic, however, to manipulate the speedometer in real time using keystrokes alone. This is because spoofed speed messages must be sent repeatedly at intervals of 10 milliseconds, otherwise the speedometer will default back to displaying the true speed being reported by the vehicle's PCM.

It was found that the target vehicle's speedometer readout could be manipulated for up to several minutes at a time. The simplest form of speedometer attack would be to simply craft one CAN 158 message for the desired speed and send that same message to the vehicle repeatedly at 10-millisecond intervals. However, the requisite "pulse" signal of the 158 message byte 8 adds an element of complexity. In order for the Gauge Control Module (GCM) to process and display the spoofed speed message, it first has to be convinced that the vehicle is actually moving. This was accomplished by copying the legitimate byte 8 signal recorded during an actual driving session, overlaid with the spoofed speed data of bytes 5 & 6. An excerpt from the final speedometer manipulation .log file is shown below in *Figure 29*:

```
(1492993910.700000)  slcan0     158#0000000076300031
(1492993910.710000)  slcan0     158#0000000076300009
(1492993910.720000)  slcan0     158#000000007630001B
(1492993910.730000)  slcan0     158#0000000076300020
```

roderick.h.currie@gmail.com

```
(1492993910.740000)  slcan0  158#000000007630003C
(1492993910.750000)  slcan0  158#000000007630000F
(1492993910.760000)  slcan0  158#000000007630001E
(1492993910.770000)  slcan0  158#000000007630002B
(1492993910.780000)  slcan0  158#0000000076300032
(1492993910.790000)  slcan0  158#0000000076300005
(1492993910.800000)  slcan0  158#0000000076300016
(1492993910.810000)  slcan0  158#0000000076300023
(1492993910.820000)  slcan0  158#0000000076300038
```

***Figure 29*: Excerpt from Speedometer Attack .log File**

In the .log file excerpt shown above, bytes 1 through 4 have been set to zero because they are not required for manipulation of the speedometer. Bytes 5 & 6 contain the hexadecimal value 7630, which equates to a speed of 188 mph. Byte 7 has also been set to zero as it is not required for this exercise. Finally, the changing data in the byte 8 position represents the in-motion "pulse" signal.

The photograph shown below in *Figure 30* shows an uncommon sight – a reading of 188 mph being displayed on the speedometer while the vehicle itself is stationary. This represents a successful attack against the target vehicle's speedometer. This also points to a worrisome lack of checks and balances on the CAN bus, allowing a would-be attacker to create a condition which falls outside of "normal" operating parameters without any intervention from the vehicle.

***Figure 30*: Successful Attack on Speedometer**

roderick.h.currie@gmail.com

A video of the speedometer attack from this project is also available online at the following URL: https://www.youtube.com/watch?v=QlpTx_LsW7M

23. Manipulating Odometer Data

The odometer displays the number of miles a vehicle has traveled in its lifetime. It should be noted that modifying a vehicle's odometer is illegal in the United States under Title 49, U.S. Code Chapter 327, which prohibits the "disconnection, resetting, or alteration of a motor vehicle's odometer with intent to change the number of miles indicated thereon" (NHTSA, 2017). Nonetheless, in the interest of security research, what follows is a method for manipulating the odometer through reverse engineering of CAN bus messages.

The way in which the odometer value is managed by the vehicle varies from one manufacturer to the next. On some vehicles, the actual odometer value in miles or kilometers is broadcast constantly on the CAN bus. Vehicles using this method are most susceptible to real-time odometer spoofing. However, the 2011 Honda Civic actually stores the odometer value within the gauge cluster itself. This is evidenced by cases in which the gauge cluster is replaced and an entirely different odometer reading is inherited by the vehicle. Therefore, on the 2011 Honda Civic, the odometer value in miles or kilometers is *not* transmitted over the CAN bus. Instead, the PCM transmits a periodic CAN signal to the Gauge Control Module to increment the odometer. The frequency of the "increment" command depends upon the traveling speed of the vehicle. The GCM listens for this signal and increments the odometer each time an "increment" command is received. This means it is still possible for an attacker to manipulate the odometer to some degree through reverse engineering of CAN messages.

As noted earlier, the odometer signal can be found in CAN message ID 158, the same message type responsible for speed and other engine data:

```
158#03BE03D803C7022C
```

More specifically, byte 7 is used for the signal to tell the odometer to periodically increment depending upon how fast the vehicle is traveling. When the vehicle is first turned on, CAN message 158 will have a value of 00 in the byte 7 position. As the

vehicle begins to move, byte 7 will change to 01, then to 02, and so on. Each time the Gauge Control Module sees an increment of the byte 7 value, it, in turn, increments the stored odometer value. The actual numeric value of byte 7 is insignificant – it is only the timing of the increments that the Gauge Control Module cares about. Also, the fact that one byte of data can only cycle through 256 possible values is of little consequence; when byte 7 reaches a decimal value of 255, its next increment returns it to a value of zero and the incrementation cycle continues. The graph shown below in *Figure 31* overlays the byte 7 odometer signal on top of vehicle speed data from bytes 5 & 6:

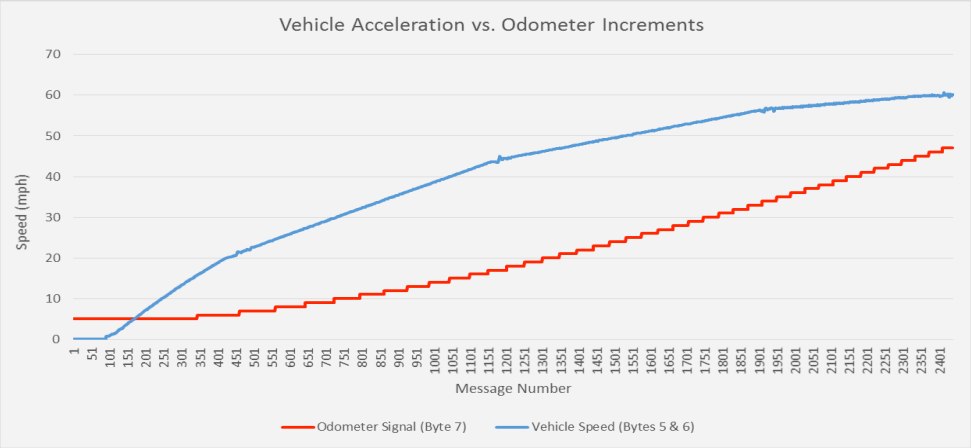

***Figure 31*: Relationship of Vehicle Speed and Odometer Incrementation**

The graph above was created using data recorded while accelerating the vehicle from a stop to a speed of approximately 60 mph. What is immediately apparent from the graph is that as the traveling speed of the vehicle increases, so too does the frequency of odometer incrementation. When the vehicle is traveling at a low speed, the "steps" of the odometer line appear more spread out. As the vehicle travels faster, the time between each odometer increment lessens.

It became apparent that the key to manipulating the odometer is to duplicate the odometer incrementation signal. And, in theory, doing so with a minimal length of time between increments should cause the odometer readout to increase more rapidly. Using lessons learned in the speedometer manipulation exercise, it was also found that the byte 8 vehicle-in-motion "pulse" signal had to be present to get any kind of response from the odometer.

roderick.h.currie@gmail.com

As with the tachometer and speedometer hacks, the easiest way to manipulate the vehicle's odometer is to play back a custom .log file using canplayer. This is because the Gauge Control Module expects to receive a CAN 158 message from the Powertrain Control Module once every 10 milliseconds, without fail. An excerpt from the .log file used to manipulate the odometer is shown below in *Figure 32*:

```
(1492993910.840000)  slcan0  158#0000000000003508
(1492993910.850000)  slcan0  158#000000000000351E
(1492993910.860000)  slcan0  158#000000000000352C
(1492993910.870000)  slcan0  158#000000000000353C
(1492993910.880000)  slcan0  158#0000000000003506
(1492993910.890000)  slcan0  158#0000000000003511
(1492993910.900000)  slcan0  158#0000000000003624
(1492993910.910000)  slcan0  158#0000000000003636
(1492993910.920000)  slcan0  158#000000000000360D
(1492993910.930000)  slcan0  158#0000000000003614
(1492993910.940000)  slcan0  158#0000000000003627
(1492993910.950000)  slcan0  158#000000000000363B
(1492993910.960000)  slcan0  158#0000000000003601
(1492993910.970000)  slcan0  158#0000000000003615
(1492993910.980000)  slcan0  158#000000000000372B
(1492993910.990000)  slcan0  158#000000000000373B
```

***Figure 32*: Excerpt from Odometer Attack .log File**

As can be seen from the excerpt above, only bytes 7 and 8 of the 158 message are needed to increase the vehicle's odometer value. Byte 8, highlighted in purple, is the "pulse" signal tricking the vehicle into thinking it is in motion. Byte 7, highlighted in red, is the byte that actually increments the odometer. In order to achieve a rapid incrementation of the odometer, it was determined that the optimal span between each increment of the byte 7 value was 8 messages, or a time interval of 80ms. When the byte 7 value was incremented more frequently than once every 80ms, some of the increment requests were actually ignored and the odometer readout would increment at a much slower rate.

When the above .log file was played back to the vehicle continuously for a period of 2 minutes, the odometer increased by a total of 5.7 miles. Under normal circumstances,

roderick.h.currie@gmail.com

such a rapid rate of odometer increase would require the vehicle to be traveling at a speed of approximately 171 miles per hour.

Modification of the odometer is not something that can be captured in a photograph. However, a video of the odometer attack from this project is available online at the following URL: https://www.youtube.com/watch?v=YhKfDh1-KP4

24. Implications

The specific attacks demonstrated in this paper may seem insignificant, as they do not directly impact the safety or control of the vehicle. However, the intent of this project was to demonstrate the relative ease with which an unauthorized device can join the CAN bus and manipulate the vehicle. Controlling the data being displayed on the vehicle's gauge cluster is an effective way to demonstrate manipulation of the CAN bus, as it provides a clear, visual indicator of a successful attack. This is similar to the way a hacker might breach a web server and deface a website as evidence of their exploits.

What is particularly alarming about this research is that the same network bus that was exploited to manipulate the gauge cluster also hosts communications for the engine, transmission, brakes, and steering. On a newer, more connected vehicle with a greater amount of computerization, the same techniques demonstrated in this paper could be used to take full control of the vehicle and create a dangerous, potentially deadly situation.

25. Conclusion

By successfully demonstrating multiple ways in which the CAN bus can be manipulated using basic computer hardware, this project has highlighted just how woefully insecure the CAN architecture is. Unfortunately, without a revision of the federal law mandating the OBD-II standard, the outdated CAN bus is not going away anytime soon. The Controller Area Network will remain at the core of modern vehicles for years to come.

Today, with the auto industry on the cusp of fully-autonomous vehicle technology and greater interconnectivity than ever before, auto manufacturers simply cannot ignore the inherent vulnerabilities of the CAN bus. The problem of securing automotive systems

roderick.h.currie@gmail.com

is massively complex, but the responsibility lies with the automakers to face the challenge head-on and take proactive steps to secure their products. In the meantime, the best way to provoke the automakers to action is to increase public awareness of the underlying problem so that it can no longer be ignored. It is my hope, therefore, that the research presented in this paper will inspire other security researchers to undertake car hacking projects of their own for the betterment of the automobile industry overall.

roderick.h.currie@gmail.com

Appendix A

List of CAN Messages and Functions for 2011 Honda Civic LX

CAN ID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Vehicle Function
13A	XX	-	-	-	-	-	-	-	Engine Idle Compensation
13A	-	XX	-	-	-	-	-	-	Gas Pedal Position
158	XX	XX	-	-	-	-	-	-	Speed Data, Not for Speedometer
158	-	-	XX	XX	-	-	-	-	Engine RPM, Not for Tachometer
158	-	-	-	-	XX	XX	-	-	Speed Data for Speedometer
158	-	-	-	-	-	-	XX	-	Odometer Incrementation
158	-	-	-	-	-	-	-	XX	Vehicle in Motion
164	00	-	-	-	-	-	-	-	Headlights: Off
164	04	-	-	-	-	-	-	-	Headlights: DRLs
164	05	-	-	-	-	-	-	-	Headlights: Parking Lights
164	06	-	-	-	-	-	-	-	Headlights: Low Beams
164	07	-	-	-	-	-	-	-	Headlights: High Beams
164	-	-	00	-	-	-	-	-	A/C: Off (Engine Running)
164	-	-	40	-	-	-	-	-	A/C: On (Engine Running)
164	-	-	80	-	-	-	-	-	A/C: Off (Engine Off)
164	-	-	C0	-	-	-	-	-	A/C: On (Engine Off)
17C	-	-	XX	XX	-	-	-	-	Engine RPM, Not for Tachometer
17C	-	-	-	-	XX	-	XX	-	Brake Pedal Depressed
1DC	02	XX	XX	XX					Engine RPM for Tachometer
188	-	-	-	01	-	-			Current Gear: Park
188	-	-	-	02	-	-			Current Gear: Reverse
188	-	-	-	04	-	-			Current Gear: Neutral
188	-	-	-	08	-	-			Current Gear: Drive
188	-	-	-	20	-	-			Current Gear: 3rd
188	-	-	-	40	-	-			Current Gear: 2nd
188	-	-	-	80	-	-			Current Gear: 1st
164	0X	-	-	-	-	-	-	-	Cruise Control: Off
164	2X	-	-	-	-	-	-	-	Cruise Control: On
164	AX	-	-	-	-	-	-	-	Cruise Control: Accelerate
164	6X	-	-	-	-	-	-	-	Cruise Control: Decelerate
164	EX	-	-	-	-	-	-	-	Cruise Control: Cancel
164	X4	-	-	-	-	-	-	-	Handbrake: On
164	X0	-	-	-	-	-	-	-	Handbrake: Off
294	04	-	-	-	-	-	-	-	Turn Signals: Off
294	24	-	-	-	-	-	-	-	Turn Signals: Left
294	44	-	-	-	-	-	-	-	Turn Signals: Right
294	04	-	-	-	-	-	-	-	Wipers: Off
294	0C	-	-	-	-	-	-	-	Wipers: Intermittent
294	14	-	-	-	-	-	-	-	Wipers: Low
294	1C	-	-	-	-	-	-	-	Wipers: High
305	80	-							Driver's Seatbelt: Unfastened
305	00	-							Driver's Seatbelt: Fastened
324	XX	XX	XX	XX	XX	XX	XX	XX	Engine Run Time Clock
40C	XX	XX	XX	XX	XX	XX	XX	XX	Vehicle Identification Number (VIN)

roderick.h.currie@gmail.com

References

ALLDATA. (2017). *ALLDATA – OEM Repair Information for Professionals*. Retrieved March 31, 2017, from http://www.alldata.com/

Amazon. (2015). *AmazonBasics USB 2.0 Cable - A-Male to B-Male - 6 Feet (1.8 Meters)*. Retrieved March 30, 2017, from https://www.amazon.com/gp/product/B00NH11KIK/

B&B Electronics. (2011). *Does My Car Have OBD-II?* Retrieved March 25, 2017, from http://www.obdii.com/connector.html

Canonical. (2011). *Ubuntu on Lenovo Thinkpad T420s*. Retrieved March 30, 2017, from https://certification.ubuntu.com/hardware/201102-7326/

CERN. (2013). *The Birth of the Web*. Retrieved May 3, 2017, from https://home.cern/topics/birth-web

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S. et al. (2011). *Comprehensive Experimental Analyses of Automotive Attack Surfaces.* Retrieved March 25, 2017, from http://www.autosec.org/pubs/carsusenixsec2011.pdf

Currie, R. (2015). *Developments in Car Hacking*. Retrieved March 17, 2017, from https://www.sans.org/reading-room/whitepapers/ICS/developments-car-hacking-36607

Currie, R. (2016). *The Automotive Top 5: Applying the Critical Controls to the Modern Automobile*. Retrieved March 17, 2017, from https://www.sans.org/reading-room/whitepapers/critical/automotive-top-5-applying-critical-controls-modern-automobile-36862

EpochConverter. (2017). *Epoch & Unix Timestamp Conversion Tools*. Retrieved April 25, 2017, from https://www.epochconverter.com/

Evenchick, E. (2015). *CANtact: The Open Source Car Tool*. Retrieved March 30, 2017, from http://linklayer.github.io/cantact/

Gallagher, S. (2015). *Researchers reveal electronic car lock hack after 2-year injunction by Volkswagen*. Retrieved March 23, 2017, from https://arstechnica.com/security/2015/08/researchers-reveal-electronic-car-lock-hack-after-2-year-injunction-by-volkswagen/

roderick.h.currie@gmail.com

Golson, J. (2016). Car hackers demonstrate wireless attack on Tesla Model S. Retrieved February 10, 2017, from http://www.theverge.com/2016/9/19/12985120/tesla-model-s-hack-vulnerability-keen-labs

Greenberg, A. (2015). *After Jeep Hack, Chrysler Recalls 1.4M Vehicles for Bug Fix*. Retrieved May 1, 2017, from https://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/

Greenberg, A. (2016). *It's Finally Legal To Hack Your Own Devices (Even Your Car)*. Retrieved March 24, 2017, from https://www.wired.com/2016/10/hacking-car-pacemaker-toaster-just-became-legal/

Greenberg, A. (2017). *Securing Driverless Cars From Hackers is Hard. Ask the Ex-Uber Guy Who Protects Them*. Retrieved May 3, 2017, from https://www.wired.com/2017/04/ubers-former-top-hacker-securing-autonomous-cars-really-hard-problem/

Hartkopp, O., & Thürmann, U. (2006). *Low Level CAN Framework*. Retrieved March 30, 2017, from https://www.brownhat.org/docs/socketcan/llcf-api.html

Kernel.org. (2016). *Readme File for the Controller Area Network Protocol Family (aka SocketCAN)*. Retrieved March 30, 2017, from https://www.kernel.org/doc/Documentation/networking/can.txt

Kleine-Budde, M. (2012). *SocketCAN - The official CAN API of the Linux kernel*. Retrieved March 31, 2017, from https://www.can-cia.org/fileadmin/resources/documents/proceedings/2012_kleine-budde.pdf

Kvaser. (2014). *Kvaser's CanKing - Free Bus Monitor Software*. Retrieved March 30, 2017, from http://www.kvaser.com/canking/

Lenovo. (2015). *Detailed specifications - ThinkPad T420*. Retrieved March 30, 2017, from https://support.lenovo.com/us/en/solutions/pd015734

Linklayer. (2016). *SocketCAN*. Retrieved March 31, 2017, from https://wiki.linklayer.com/index.php/SocketCAN

Lyons, A. (2015). *On-Board Diagnostics (OBD) Program Overview*. Retrieved March 29, 2017, from http://www.theicct.org/sites/default/files/6_ARB_OBD.pdf

Miller, C., & Valasek, C. (2014). *Adventures in Automotive Networks and Control Units.* Retrieved March 16, 2017, from http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf

roderick.h.currie@gmail.com

Miller, C., & Valasek, C. (2015). *Remote Exploitation of an Unaltered Passenger Vehicle.* Retrieved March 16, 2017, from http://illmatics.com/Remote%20Car%20Hacking.pdf

Mitchell1. (2017). *Automotive Repair Software and Repair Shop Solutions | Mitchell1*. Retrieved March 31, 2017, from http://mitchell1.com/

Nakade M. J., Shrivastava K. N., Dhumal S. B., Wakle R. P. (2015). *Communication over Automobile Instrument Assembly using CAN Bus –A Review*. Retrieved March 29, 2017, from http://www.ijmter.com/published_special_issues/07-02-2015/communication-over-automobile-instrument-assembly-using-can-bus-a-review.pdf

Netronics. (2015). *CANdo Application*. Retrieved March 30, 2017, from http://www.cananalyser.co.uk/candoapp.html

Nissanhelp. (2011). *OBDII Data Link Connector (DLC) Location*. Retrieved March 29, 2017, from http://nissanhelp.com/diy/titan/projects/nissan_titan_obd_connector_location.php

NHTSA. (2017). *Odometer Information Overview for Consumers*. Retrieved May 4, 2017, from https://one.nhtsa.gov/Vehicle-Safety/Odometer-Fraud/Odometer-Information-Overview-for-Consumers

O'Carroll, L. (2013). *Car hacking scientists agree to delay paper that could unlock Porsches*. Retrieved March 23, 2017, from https://www.theguardian.com/technology/2013/jul/30/car-hacking-ignition-injunction

O'Kane, S. (2015). *Automakers Just Lost the Battle to Stop You from Hacking Your Car*. Retrieved March 25, 2017, from http://www.theverge.com/2015/10/27/9622150/dmca-exemption-accessing-car-software

SparkFun. (2015). *OBD-II to DB9 Cable*. Retrieved March 30, 2017, from https://www.sparkfun.com/products/10087

Smith, C. (2016). *The Car Hacker's Handbook: A Guide for the Penetration Tester*. San Francisco, CA: No Starch Press.

Talbot, S., & Ren, S. (2008). *Comparison of FieldBus Systems, CAN, TTCAN, FlexRay and LIN in Passenger Vehicles*. Retrieved March 31, 2017, from http://www.

talbotsystems.com/documents/Comparision_of_FieldBus_Systems_CAN_TTCAN_FlexRay_and_LIN_in_Passenger_Vehicles.pdf

Twelfth Round Auto. (2017). *5 Best Jack Stands: A Review of the Top Products*. Retrieved May 1, 2017, from https://www.twelfthroundauto.com/best-jack-stands/

U.S. Copyright Office. (1998). *The Digital Millennium Copyright Act of 1998 - U.S. Copyright Office Summary*. Retrieved March 25, 2017, from https://www.copyright.gov/legislation/dmca.pdf

Verdult, R., Garcia, F., & Ege, B. (2015). *Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer*. Retrieved March 23, 2017, from https://www.usenix.org/sites/default/files/sec15_supplement.pdf

Walter, P. (2015). *Step-by-step guide: Installing the Lawicel CANUSB adapter on Linux*. Retrieved April 4, 2017, from http://pascal-walter.blogspot.com/2015/08/installing-lawicel-canusb-on-linux.html

Wiens, K. (2015). *We Can't Let John Deere Destroy the Very Idea of Ownership*. Retrieved March 25, 2017, from https://www.wired.com/2015/04/dmca-ownership-john-deere/

Wikipedia. (2009). *File:Socketcan.png*. Retrieved March 30, 2017, from https://en.wikipedia.org/wiki/File:Socketcan.png

Wikipedia. (2014). *File:CAN-Bus-frame in base format without stuffbits.svg*. Retrieved November 27, 2015, from https://commons.wikimedia.org/wiki/File:CAN-Bus-frame_in_base_format_without_stuffbits.svg

roderick.h.currie@gmail.com

Last Updated: May 28th, 2018

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Rocky Mountain 2018	Denver, COUS	Jun 04, 2018 - Jun 09, 2018	Live Event
SEC487: Open-Source Intel Beta Two	Denver, COUS	Jun 04, 2018 - Jun 09, 2018	Live Event
SANS London June 2018	London, GB	Jun 04, 2018 - Jun 12, 2018	Live Event
DFIR Summit & Training 2018	Austin, TXUS	Jun 07, 2018 - Jun 14, 2018	Live Event
Cloud INsecurity Summit - Washington DC	Crystal City, VAUS	Jun 08, 2018 - Jun 08, 2018	Live Event
SANS Milan June 2018	Milan, IT	Jun 11, 2018 - Jun 16, 2018	Live Event
Cloud INsecurity Summit - Austin	Austin, TXUS	Jun 11, 2018 - Jun 11, 2018	Live Event
SANS Philippines 2018	Manila, PH	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Cyber Defence Japan 2018	Tokyo, JP	Jun 18, 2018 - Jun 30, 2018	Live Event
SANS Oslo June 2018	Oslo, NO	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS ICS Europe Summit and Training 2018	Munich, DE	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Crystal City 2018	Arlington, VAUS	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Cyber Defence Canberra 2018	Canberra, AU	Jun 25, 2018 - Jul 07, 2018	Live Event
SANS Minneapolis 2018	Minneapolis, MNUS	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Paris June 2018	Paris, FR	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Vancouver 2018	Vancouver, BCCA	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS London July 2018	London, GB	Jul 02, 2018 - Jul 07, 2018	Live Event
SANS Cyber Defence Singapore 2018	Singapore, SG	Jul 09, 2018 - Jul 14, 2018	Live Event
SANS Charlotte 2018	Charlotte, NCUS	Jul 09, 2018 - Jul 14, 2018	Live Event
SANSFIRE 2018	Washington, DCUS	Jul 14, 2018 - Jul 21, 2018	Live Event
SANS Malaysia 2018	Kuala Lumpur, MY	Jul 16, 2018 - Jul 21, 2018	Live Event
SANS Pen Test Berlin 2018	Berlin, DE	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS Cyber Defence Bangalore 2018	Bangalore, IN	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS Riyadh July 2018	Riyadh, SA	Jul 28, 2018 - Aug 02, 2018	Live Event
Security Operations Summit & Training 2018	New Orleans, LAUS	Jul 30, 2018 - Aug 06, 2018	Live Event
SANS Pittsburgh 2018	Pittsburgh, PAUS	Jul 30, 2018 - Aug 04, 2018	Live Event
SANS August Sydney 2018	Sydney, AU	Aug 06, 2018 - Aug 25, 2018	Live Event
SANS San Antonio 2018	San Antonio, TXUS	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS Boston Summer 2018	Boston, MAUS	Aug 06, 2018 - Aug 11, 2018	Live Event
Security Awareness Summit & Training 2018	Charleston, SCUS	Aug 06, 2018 - Aug 15, 2018	Live Event
SANS Hyderabad 2018	Hyderabad, IN	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS New York City Summer 2018	New York City, NYUS	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Atlanta 2018	OnlineGAUS	May 29, 2018 - Jun 03, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced