Spring 2019

Security Analysis of Vehicle to Vehicle Arada Locomate On Board Unit

Ramanni J Veeraraghava
vrjs@iastate.edu

Security Analysis of Vehicle to Vehicle Arada Locomate On Board Unit

by

Sudharrshan Veeraraghava Ramanni Janaarthanan

A creative component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Lotfi Ben Othmane, Co-major Professor
Doug Jacobson, Co-major Professor

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to sincerely thank Dr. Lotfi Ben-Othmane for his constant guidance throughout the research and also Prof. Doug Jacobson for his constant support.

I would also like to thank my friends and family for their loving guidance during the writing of this work. I would additionally like to thank Geethika Singh, Ashraf Shaik Mohammed and Moataz Abdelkhalek for their help throughout the project.

I thank Dr. Nicholas Multari, Professor Ashfaq Khokhar, Professor Bharat Bhargava, and Professor Manimaran Govindarasu for supporting the work financially. My work was mainly funded by the Pacific Northwest National Laboratories through the Iowa State University Information Assurance Center. Equipment and software tools were funded by the Qatar National Research Fund and The ECpE Department Excellence fund.

ABSTRACT

Arada Locomate On-Board Unit is a vehicle-to-vehicle communication device that supports the WAVE protocol, which is the standard for vehicle to vehicle communication. Successful attacks on the device could be used to control the behavior of the connected vehicle. This creative component assesses the security of the device and discusses the vulnerabilities of the applications installed on the device. It reports about our results to exploit the known vulnerabilities of Dropbear ssh, Busybox telnet, and the linux kernel, which are installed on the device and discusses how to obtain the private keys of the device to use them for attacks. In addition, it describes our investigation of the existence of exploitable buffer overflow in the *usbd* program, which accepts messages through port 6666 (IRC port). The results are: the exploitation of Dropbear ssh, Busybox telnet failed, the exploitation of the *vmsplice* vulnerability in the Linux kernel required adapting the exploit to the MIPS architecture, there is no exploitable buffer overflow in the *usbd*; however, the private keys of the device are easily accessible and the user password of the device could be changed without authentication. The current results are not that useful to stage attacks but further work may lead to exploit the device and use it to inject messages to the connected vehicle, e.g., develop an exploit for *vmsplice* vulnerability for MIPS Linux.

CHAPTER 1. INTRODUCTION

The field of automotive electronics has been a developing field in recent times which involves various machine learning techniques, reliable communication protocols, and accurate image processing techniques integrated to give a user-friendly and self-driving autonomous car. With new innovations and new patches of code released every day, there comes security issues without the developers being aware of the code fragments that have vulnerabilities. This poses a great threat to deploy the units before rigorously testing them with regards to security.

Each manufacturer has their own interfaces and software supporting them, which can allow an attacker to meddle around different platforms and perform any attacks. Vehicle to Any (V2X) communication is also enabled, which leads to a situation where cars can operate through mobile communication standards like LTE and 5G. Also, a lot of cars are enabled to have a Over The Air (OTA) upgrades which enable a greater scenario like a man-in-the-middle attack.

1.1 Problem

Network-based architectures increase the level of accessibility to different aspects and information of the hardware. The Vehicle to Vehicle (V2V) protocol is a WiFi-based protocol that can lead to various vulnerabilities in the system. Any data sent or received can be sniffed, spoofed and that can further be used to jam the network. Also, the software used can have vulnerabilities that attackers can take advantage of. The solution for this is a standardized process that acts as an evaluation of the system entirely before being deployed in the market.The research answers the following question:

Is the V2V system ARADA Locomate On Board Unit(OBU) secure?

1.2 Approach

The goal of the research was to study the hardware and perform security testing on a V2V architecture-based device. Arada Systems is one of the companies that work on developing V2V architecture-based devices. These devices find use in the Platooning trucks, cars, etc.

As the first step, the features and the software the device supports are studied. The next step in the process is to analyze the device and develop a flaw hypothesis. Later, the hypothesis is tested using various tools and other methods that give us complete knowledge on what part of the device is vulnerable.

1.3 Hardware

The system selected for study here is Arada Locamate On Board Unit (OBU) which acts as a user side unit in the network. The OBU is based on the WAVE protocol which is a WiFi-based protocol for V2V networks. It also has support for CAN bus. Figures 1.1 & 1.2 shows the Arada Locomate On-Board Unit.

1.4 Organization

This creative component is organized into 6 chapters. After this introductory chapter, chapter 2 deals with the background required for this research and also different tools that were used in the process. Chapter 3 discusses related work with the V2V implementations. Chapter 4 describes the flaw hypothesis for Arada Locomate OBU and also the experiments conducted on the system. Chapter 5 focuses on the flaw that was detected to study in detail. Finally, the conclusion and future work are discussed in Chapter 6.

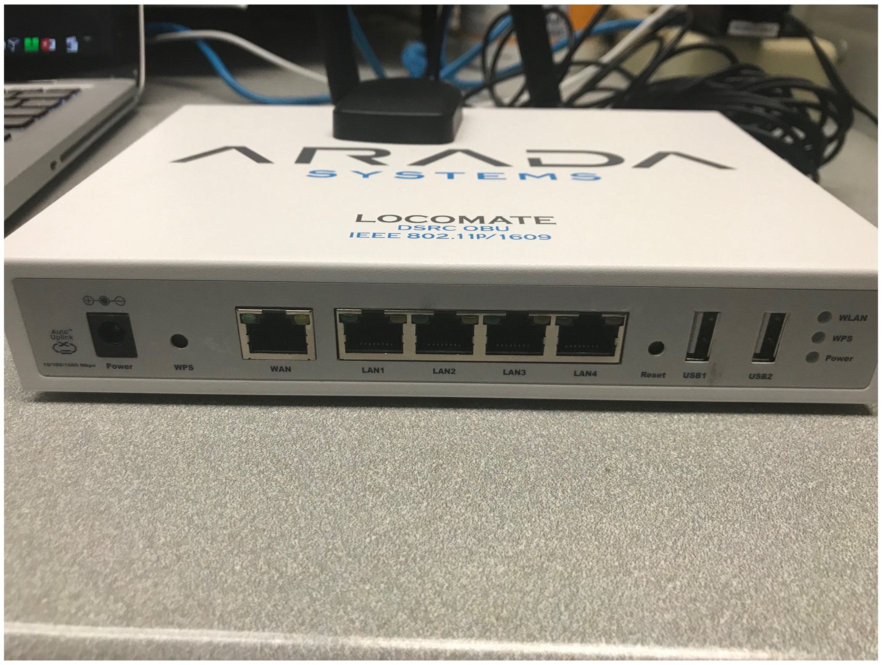


Figure 1.1 Front view of Arada Locomate OBU

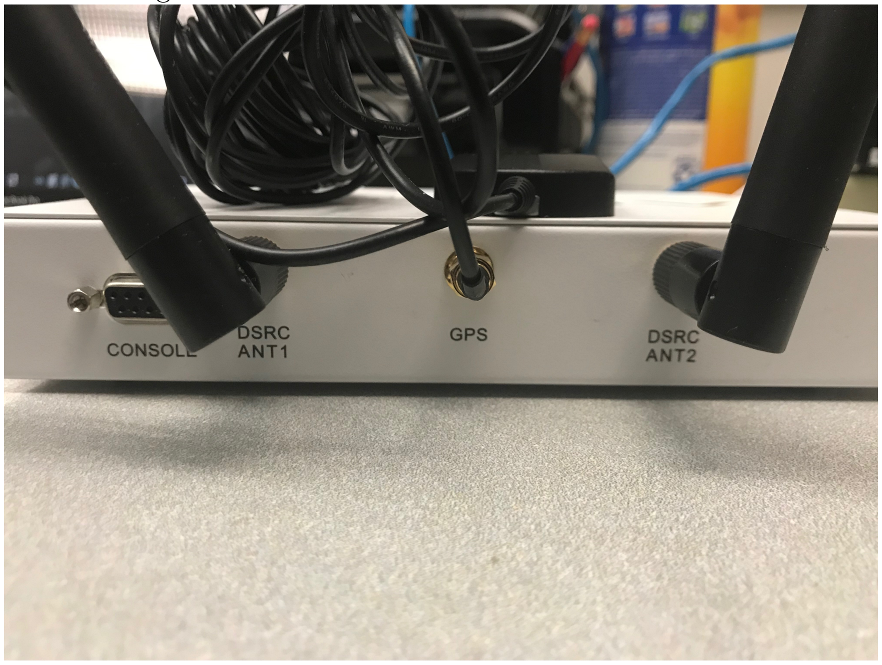


Figure 1.2 Rear view of Arada Locomate OBU

CHAPTER 2. BACKGROUND

2.1 Arada Locomate On-Board Unit

Arada Systems is a company that focuses on providing technological solutions including automotive and security. One of the products of Arada Systems for V2V communication between cars is Arada Locomate OBU. The OBU acts as the in-unit entity and the Road Side Unit (RSU) acts as the master entity collecting data from many OBU's. The RSU then processes the details and shares them with the other OBUs to coordinate among the vehicles, resulting in a safer autonomous functioning. The OBU and RSU both transfer a message called Basic Safety Message (BSM) which is essentially the critical data of the car. The BSM contains the GPS location, speed, and RPM which are also customizable by the manufacturer.

The following are the features of Arada Locomate OBU:

- Platform: Linux 2.6.23
- SDK with C libraries
- Interactive communication- SSH/Telnet
- Support for CAN bus
- WAVE protocol
- 16MB flash
- 64MB SDRAM
- Supports encryption of messages

The interfaces that the system supports are:

- USB

- Ethernet
- GPS Antenna
- DSRC Antenna
- JTAG
- Bluetooth
- WPS
- DB9 for CAN bus support

2.2 Penetration testing tools

Penetration testing tools act as an essential part of the experimenting stage. There is a broad set of tools that can help acquiring information and creating exploits. The following sections describe the tools such as Nmap, Metasploit, and Binwalk used in the project.

2.2.1 NMAP

Network Mapper (Nmap) [1] is an open source software that acts as a platform scanner and evaluates different services that run on the hosts. The features of Nmap include host discovery, port scanning, version detection, and Operating systems(OS) detection. As mentioned earlier, Nmap detects any host in the network and scan all the open ports associated with the host. It can also be used to detect the OS running on the target machine.

The usage of Nmap command is:

nmap [*options*] $IP\_Address$

The options here can range according to the output required.

In our case, we require the service running of each port and the version of the services. So the $-sV$ option is used. Also, 192.168.0.40 is the pre-set IP address of Arada Locomate OBU. So the command in our case is as follows:

$nmap\ \ -sV$ 192.168.0.40

2.2.2 Metasploit

Metasploit [2] is a security platform that offers details about various vulnerabilities in services and also provides a platform to develop exploits that can run over the network. In this project, Metasploit was used to get information about vulnerabilities of the services running on the OBU machine. Also, in the penetration testing process, it was used to test with the previously found exploits in the database.

2.2.3 Binwalk

In general, firmware images are compressed images that contain a kernel and a root file system. These are not common compression formats and are uniquely designed to satisfy the memory constraints. One of the Kali Linux-based tools used for analyzing the firmware images/executables/binaries is Binwalk which is best when firmware images are the test item.

The usage of this utility is as given below:

$binwalk\ \ [options]\ \ Firmware\_image$

The options are set in various categories as seen in [3].

In our case, the root file system had to be extracted that demanded Binwalk to use file extraction options. The compressed file used in our case was the rootsqaushfs file. So the command in our case is:

$binwalk\ \ -e\ \ rootfilesystem.squashfs$

where e specifies the Binwalk to extract the file in the current location.

2.3 Reverse Engineering tools

2.3.1 Radare2

Radare2 [4] is an open source framework used for reverse engineering, debugging, and analyzing binaries. A GUI version is also available. Radare2 supports various formats of files like ELF, MZ, Java class, and file systems. It also supports a broad range of instruction sets like Intel x86, MIPS, and ARM. Radare2 supports static analysis including disassembling the binary, extracting information from symbols, and also dynamic analysis including support for a built-in low-level debugger. In this project, Radare2 is used to acquire the basic understanding of the program and to reassure the definition of port 6666 through the call graph.

The command to call radare2 is either r2 or radare2 with the options and executable path.

$radare2\ \ [-options]\ \ /path/to/binary$

Various options are present for different functionality. The project requires the disassembly of the executable and obtain the call-graph which demands the usage of the following command:

Command 1: $r2\ \ /path/to/binary$

Command 2: $agf\ >\ output.dot$

Here the flags agf in command 2 indicates to disassemble, output the graphs into the file specified.

Radare2 uses Graph viz tool for constructing call graphs. The supported format is .dot which requires another step to convert to pdf as follows:

$dot\ \ -Tpdf\ \ output.dot\ \ -o\ \ output.pdf$

2.3.2 IDA Pro

Interactive Disassembler (IDA)Pro [5] is a popular proprietary disassembler used for program dis-assembly and debugging binaries. It is a GUI-based platform and is designed to provide user-friendly functionality. It supports Windows, Linux, and Mac OS X hosts. The recognized file formats include ELF, MZ, and Mach-O. It supports a wide range of instruction sets including Intel x86, MIPS, and ARM. It also supports compilers like Borland C, Microsoft C, and GNU C++. Important features of IDA include automatic code analysis and knowledge of API call parameters. In this project, IDA Pro was used to study the entire program and also to observe all the functions tied to the usbd program.

Usage of the software is straightforward. Select the file to disassemble once the IDA pro application starts. As a result, the IDA view tabs open. The main view includes the disassembled code. A user can switch between Graph view to view the call graph. There is an option for the Hex view as well. The other windows include Imports, Names, Functions, Strings, Stack view, etc. In this project, graphical view, functions and stack views were the important views used to analyze the program.

2.4 Debugging tools

2.4.1 QEMU Emulator

Quick Emulator (QEMU) [6] is an emulator to provide hardware virtualization functionality, and it is an open source software [7]. It supports a broad range of operating systems and also a broad range of instruction set architectures. It also has an option of saving and restoring the machine at the current state. The architectures supported by QEMU are x86, MIPS, SPARC, Microblaze, etc.

In this project, Linux OS is emulated in a MIPS based kernel system. The command for using QEMU through command line is:

$qemu - system - mips \;\; -M \;\; malta \;\; -kernel \;\; vmlinux - 3.2.0 - 4 - 4kc - malta \;\; -$
$hda \;\; debian\_wheezy\_mips\_standard.qcow2 \;\; -append \;\; "root = /dev/sda1" \;\; -nographic$

Here the qemu-system-mips instructs the QEMU to use MIPS architecture for the instruction set. Also the -kernel allows us to specify the kernel image. The -hda enables us to define the root file system that the QEMU will have upon starting the emulator. We can also open ports for communication using $-redir$ flag in the above command.

2.4.2 Cross Compilation

Software can be cross-compiled for any architecture. Cross compilation is necessary if a user wants to compile any code fragment for another architecture from the availability of system with different architecture. The -target variable points to the required architecture in order to perform the cross-compilation.

In this project,GDB debugger [8] is cross-compiled for the hardware that is MIPS based. The cross compilation is done through the configure file. The commands for successful cross compilation are:

Command 1: $exportLDFLAGS = static$

Command 2: $./configure\ \ --target = mips - linux - gnu\ \ CC = "mips - linux - gnu - gcc"\ \ CXX = ""mips - linux - gnu - gcc"$

Command 3: $make$

Here LDFLAGS is set as static which builds a GDB executable with statically linked libraries. –target specifies the required architecture and the CC and CXX flags instructs the compiler to use MIPS based libraries. Once the make is successful, the GDB folder contains the executable to transfer to the target device.

However, the cross-compiled versions of GDB did not work in the hardware as the OS is a proprietary version of Linux which may restrict the usage of cross-compiled versions. The figures 2.1 & 2.2 shows the two GDB cross compiled versions that failed to execute.

2.4.3 GDB

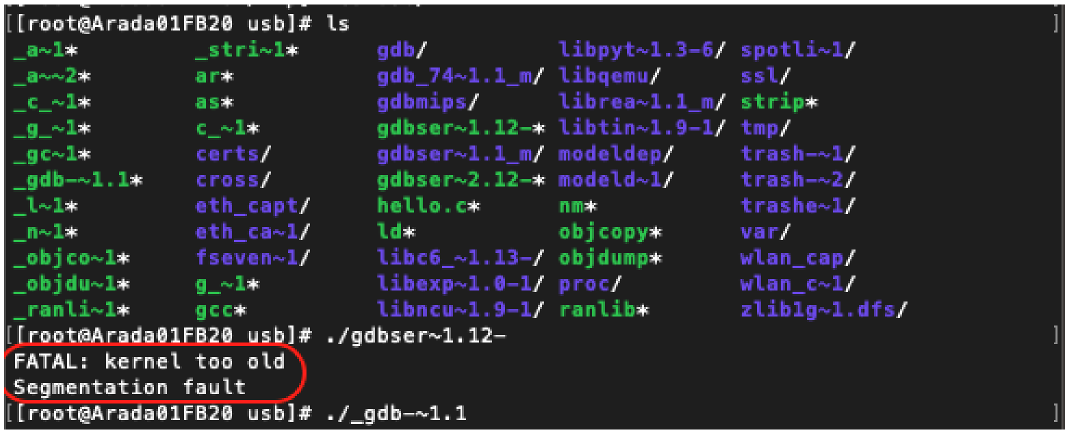
GNU Debugger (GDB) is available for various platforms and also supports debugging for various architectures. In this project, GDB was used to debug the code for any overflow in the program stack due to the presence of buffer overflow vulnerability. The breakpoints are set near the function that was suspected to have a buffer overflow.

The following commands show how to start gdb and set a breakpoint. Chapter 5 discusses further steps to analyze the program stack.

Command 1: $gdb\ \ usbd$

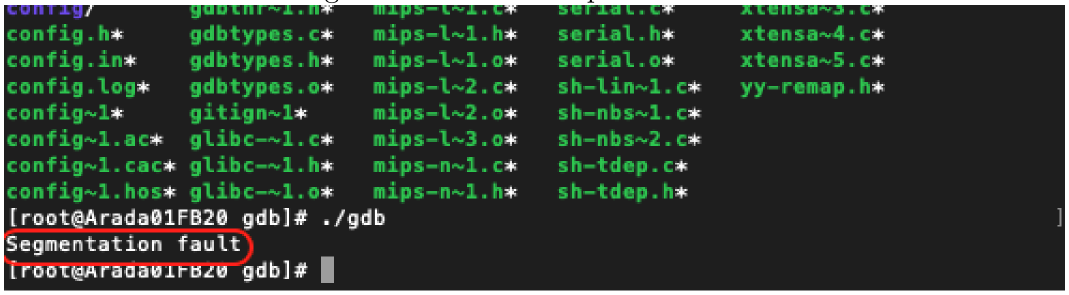
Command 2: $break\ \ read()$

Command 3: run



```
[[root@Arada01FB20 usb]# ls
 _a~1*          _stri~1*       gdb/           libpyt~1.3-6/ spotli~1/
 _a~~2*         ar*            gdb_74~1.1_m/  libqemu/      ssl/
 _c_~1*         as*            gdbmips/       librea~1.1_m/ strip*
 _g_~1*         c_~1*          gdbser~1.12-*  libtin~1.9-1/ tmp/
 _gc~1*         certs/         gdbser~1.1_m/  modeldep/     trash-~1/
 _gdb-~1.1*     cross/         gdbser~2.12-*  modeld~1/     trash-~2/
 _l~1*          eth_capt/      hello.c*       nm*           trashe~1/
 _n~1*          eth_ca~1/      ld*            objcopy*      var/
 _objco~1*      fseven~1/      libc6_~1.13-/  objdump*      wlan_cap/
 _objdu~1*      g_~1*          libexp~1.0-1/  proc/         wlan_c~1/
 _ranli~1*      gcc*           libncu~1.9-1/  ranlib*       zlib1g~1.dfs/
[[root@Arada01FB20 usb]# ./gdbser~1.12-
 FATAL: kernel too old
 Segmentation fault
[[root@Arada01FB20 usb]# ./_gdb-~1.1
```

Figure 2.1 Cross compiled version 1



```
config/        gdbthr~1.h*   mips-l~1.c*   serial.c*     xtensa~3.c*
config.h*      gdbtypes.c*   mips-l~1.h*   serial.h*     xtensa~4.c*
config.in*     gdbtypes.h*   mips-l~1.o*   serial.o*     xtensa~5.c*
config.log*    gdbtypes.o*   mips-l~2.c*   sh-lin~1.c*   yy-remap.h*
config~1*      gitign~1*     mips-l~2.o*   sh-nbs~1.c*
config~1.ac*   glibc-~1.c*   mips-l~3.o*   sh-nbs~2.c*
config~1.cac*  glibc-~1.h*   mips-n~1.c*   sh-tdep.c*
config~1.hos*  glibc-~1.o*   mips-n~1.h*   sh-tdep.h*
[root@Arada01FB20 gdb]# ./gdb
Segmentation fault
[root@Arada01FB20 gdb]#
```

Figure 2.2 Cross compiled version 2

Command 4: *c*

The gdb command followed by the program runs gdb bound with the program. Then the breakpoints are set using *break* or *b* followed by function name. Usage of *run* command will execute the program until it reaches read() and stops. Type the command *continue* or *c* to continue program execution. In a program, there is no limit on the number of breakpoints.

CHAPTER 3. RELATED WORK

This chapter discusses the current and previously existing research on security towards automotive communication protocols. The first section talks about the use of telematics and its importance in automobiles. The second section discusses how vulnerable Controller Area Network(CAN) is to attacks. It also explains the steps that can be taken to prevent the attacks against it. Finally, the third section discusses the attacks and vulnerabilities due to modern technologies used in modern vehicles.

3.1 Telematics in modern vehicles

As emphasized in [12] by Yilin Zhao, the modern society is rapidly growing with the usage of advanced technologies like Telematics. Telematics is the use of telecommunication to receive and send data over to different places. Since telecommunication is a standard for communication, the automotive sector has adapted to the usage of these technologies to improve user experience and safety. One of the best examples of telecommunication in automobiles is the *mayday* system which connects the vehicle to the nearby roadside services during an emergency as emphasized in [12]. The *mayday* system shares the location of the vehicle through the Global Positioning System(GPS) giving the exact location of the vehicle to roadside assistance teams. The entire process uses cellular communication to send and receive data over the internet or through the Short Messaging Service(SMS). Although this increases the user safety, it increases the risks too. Modern cars also use WiFi for vehicle-to-vehicle communication which brings in possibilities to take control of the network. Usage of LTE for current automotive is also under development which again raises security issues. Incorporating telecommunications in automotive is one of the best technological advancements, but it has the possibility of security breaches.

3.2 CAN bus security

Controller Area Network(CAN) bus is the brain of automobiles that contains all the important data communicated between various devices in the car. The architecture of CAN bus is such that, there are no encryption policies that can shield the CAN bus from being accessed. This poses a major threat if an attacker has access to the car. An attacker can sniff and spoof the CAN diagnostic packets or modify the firmware in the Electronic Control Units(ECUs) to acquire control over the vehicle units [13]. Various methods and steps can be followed to take control over the CAN network one of which is reverse engineering the CAN bus data and use tools to replay the manipulated data into the CAN bus [14], although the process requires physical access to the CAN bus. Two methods to safeguard the system from those types of attacks are the intrusion detection technique which stand out to be one of the best to detect,study the pattern of attacks, and create a detection method to prevent attacks. The second method is the forensic support which also proves to be one of the best [15]. Denial of service attacks is a common method followed in any network based devices and CAN is also vulnerable to these attacks. Also, reverse engineering a CAN bus or a firmware gives us ample access to take control over the vehicle [16]. In recent times, the use of various interfaces with CAN data will also increase the threat in vehicular network access demanding a study of every software in the car before deployment.

3.3 Security analysis of modern automotive technologies

With the increase of connectivity in vehicles, the increase in security threats also can be observed. In 2016, the experiments on Teslas Model S shows the ability to control various functions of the car from opening trunk to activating brakes while the vehicle is in motion. The attack was possible due to a vulnerability found in the web browser used in Teslas firmware and when connected to a malicious WiFi hotspot, the attacker was able to take control of the vehicle [17]. Alternatively, an infotainment system can give access to the cars critical data and a proprietary mobile application developer can gain access to the CAN bus thereby taking control of the system. The possibilities of heap overflows vulnerabilities in these applications allows the attacker to gain remote access thereby causing a significant threat in these applications [18]. The buffer overflow attacks through a Bluetooth interface poses an important threat as there is a possibility of crashing the system and obtain control of the vehicle. There are various types of buffer overflow like the presence of *strcpy* without proper size check which leads to a critical security threat [19]. This discussion summarises various risks in the modern automobile network and the emphasising the need for countermeasures.

CHAPTER 4. PENETRATION TESTING OF ARADA LOCOMATE

This chapter discusses the security testing in the Arada Locomate V2V hardware. The introduction presents the open ports and the versions of software running on the system. The hypothesis section discusses the flaw hypothesis based on the versions of the software and the other potential flaws of the device. Finally, the experiment sections discuss the different flaws in detail and the steps to test the hypothesis.

4.1 Introduction

Penetration testing has become an essential part of testing the products. It provides an insight into the technical flaws in the development especially in regards to security. The main tools used here are Nmap for port analysis, Metasploit for exploits, Binwalk for firmware analysis.

Nmap is a Linux based tool which provides information about the open ports in a system. It also helps in acquiring details about the software that runs in those ports. The Nmap also can check the OS running in the target device (Linux 2.6 in Arada Locomate OBU) and provides us with complete information about the system. Chapter 2 discusses the commands to use Nmap. Table 4.1 shows the output of Nmap. The flaws possible with these versions can be searched in the Common Vulnerabilities and Exposures website [9]. The next section discusses the potential flaws in the system.

Table 4.1 Open ports of Arada Locomate.

Port No.	TCP/UDP	Type	Program	Version
22	TCP	SSH	Dropbear	0.51
23	TCP	Telnet	Busybox	1.11
6666	TCP	IRC	Refer Sec. 4.3.4	Refer Sec. 4.3.4

Table 4.2 Flaw Hypothesis for programs in Arada Locomate.

Program	Version	CVE No.	Description
Linux Kernel	2.6.23	CVE-2008-0600	Linux kernel syscall vmsplice vulnerability which allows privilege escalation
Dropbear SSH	0.51	CVE-2016-3116	Arbitrary xauth command injection to gain read/write access
Busybox Telnet	1.11	N/A	Telnet crashes when a long string is sent for Username and password while logging in
USB Daemon	N/A	N/A	Open port can have a buffer overflow vulnerability

4.2 Flaw Hypothesis

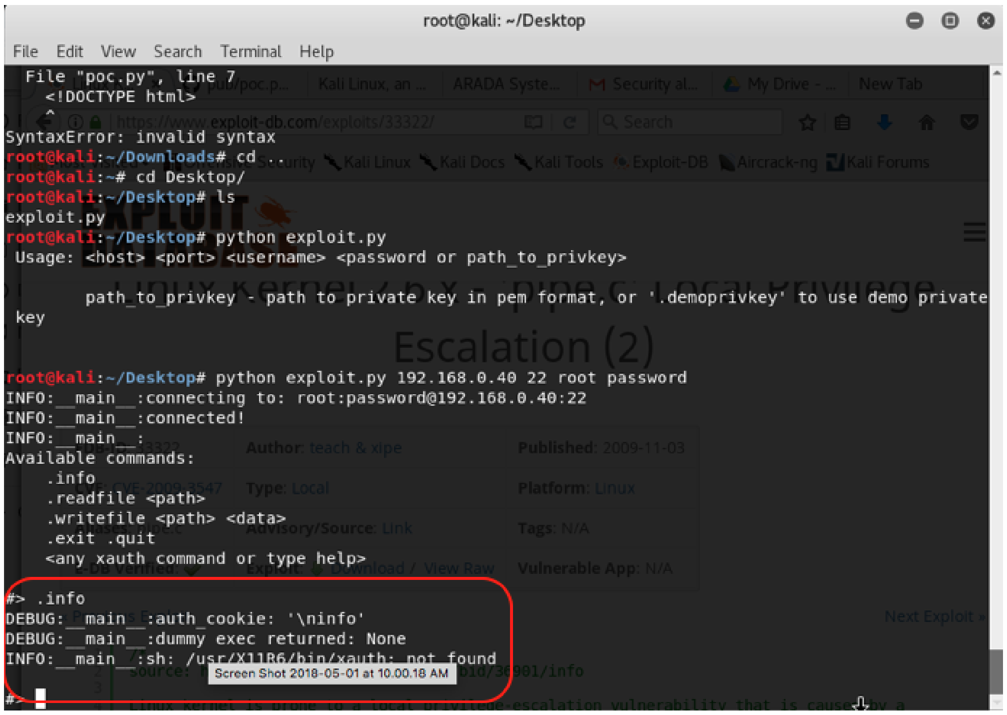
This step is the third step in the penetration testing. It involves studying the versions of software and define hypothesis of possible flaws in the device. The table 4.2 defines the hypothesis found in each software. The next section addresses the vulnerabilities in detail and presents the conducted experiments.

4.3 Experiments

4.3.1 Dropbear Xauth vulnerability

Dropbear is the SSH client and server side software. It is an open source software that supports Linux platform especially Embedded Linux. It finds use in router systems and ethernet switches. It supports xauth, which is used to acquire information from the X server. The X server extracts data from one machine to another. The vulnerability is in the case when an unauthenticated user can perform these actions. From xauth command, the attacker can get information about the environment, write file,connect to another port, and run exploits.

The Metasploit tool contains an exploit for this vulnerability. This exploit was used on the hardware as shown in Figure 4.1. It returned the status of Xauth not found. Also in firmware analysis described at a further section, there were no configuration files for Xauth which proves that the Xauth is not present. Thus the system is not vulnerable to the Dropbear Xauth vulnerability.


root@kali: ~/Desktop
File Edit View Search Terminal Help
Screen Shot 2018-05-01 at 10.00.18 AM

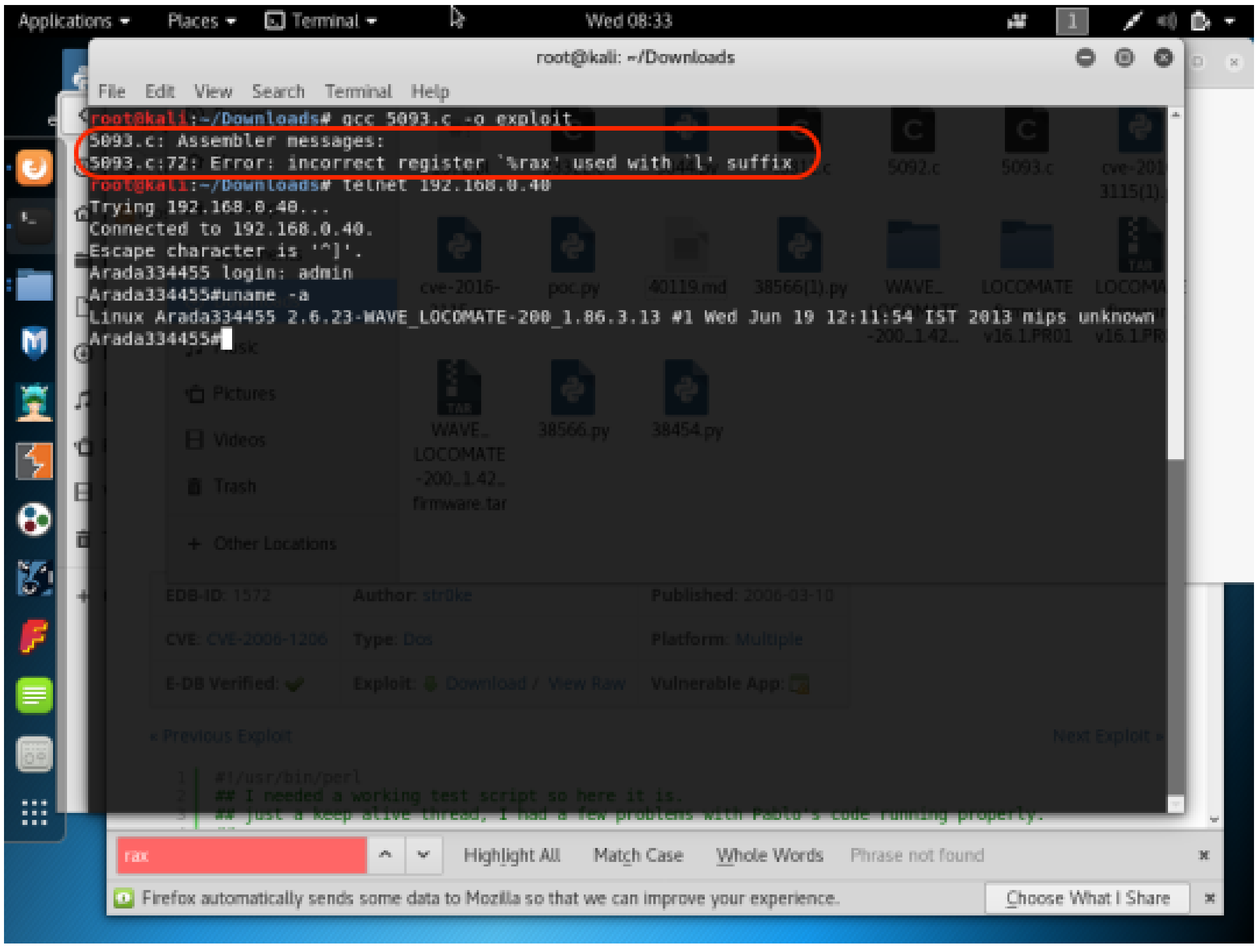
```
 File "poc.py", line 7
   <!DOCTYPE html>
   ^
SyntaxError: invalid syntax
root@kali:~/Downloads# cd ..
root@kali:~# cd Desktop/
root@kali:~/Desktop# ls
exploit.py
root@kali:~/Desktop# python exploit.py
 Usage: <host> <port> <username> <password or path_to_privkey>

        path_to_privkey - path to private key in pem format, or '.demoprivkey' to use demo private
 key

root@kali:~/Desktop# python exploit.py 192.168.0.40 22 root password
INFO:__main__:connecting to: root:password@192.168.0.40:22
INFO:__main__:connected!
INFO:__main__:
Available commands:
    .info
    .readfile <path>
    .writefile <path> <data>
    .exit .quit
    <any xauth command or type help>

#> .info
DEBUG:__main__:auth_cookie: '\ninfo'
DEBUG:__main__:dummy exec returned: None
INFO:__main__:sh: /usr/X11R6/bin/xauth: not found

#> 
```

Figure 4.1 Dropbear Xauth exploit

4.3.2 Vmsplice privilege escalation

Linux Kernels are an integral part of any computer that works with Linux OS. The kernel level operations should be highly secure as any vulnerability here can cause an entire breach of the system, and an attacker can take full access over the system. This vulnerability deals with a system call that Linux kernel version 2.6 supports [10]. Arada Locomate has a proprietary variant of *Linux* 2.6.23 as seen from the output of *Nmap*.

The vulnerability is with the *splice*() system call which helps in data flow plumbing within the kernel. Data flow plumbing is a process which provides easy and efficient management of data, data processing and storing. This helps in the increased ease of data access. The *splice*() system calls take two file descriptors as input one being the source and the other destination. It copies data from a source file to a destination file and splices it together. This functionality will essentially allow writing a trivial code or information into the destination file. A variant of *splice*() is *vmsplice*() system call, which does the same process, but the destination here is memory used by



```
root@kali:~/Downloads# gcc 5093.c -o exploit
5093.c: Assembler messages:
5093.c:72: Error: incorrect register `%rax' used with `l' suffix
root@kali:~/Downloads# telnet 192.168.0.40
Trying 192.168.0.40...
Connected to 192.168.0.40.
Escape character is '^]'.
Arada334455 login: admin
Arada334455#uname -a
Linux Arada334455 2.6.23-WAVE_LOCOMATE-200_1.86.3.13 #1 Wed Jun 19 12:11:54 IST 2013 mips unknown
Arada334455#
```

Figure 4.2 Linux vmsplice() exploit

the program. The vulnerability was found in this system call where the *vmsplice*() missed checking the permissions of the source file. Because of this, any user can indeed write to the memory through the pipe that the kernel copies and writes to the memory which is a huge path for an attacker to take control over the system.

Also, there was another issue with the same system call that is the reverse of the situation discussed above. When the source is the memory, and the destination is a file with *vmsplice*() system call, the kernel works by reading the *iovec* structures which describe the memory range of the source. Here, the *vmsplice*() omitted checks on whether the memory was readable which allows attackers to get information about the memory and the system on the whole. An exploit for this is present for Intel x86 architecture based machines[20]. Figure 4.2 shows the exploits tried in the initial experiments which did not give desirable results. The firmware analysis discussed in further sections revealed that the system is MIPS architecture based which will require the exploit to be modified accordingly. This exploit, when revised for MIPS architecture, will be one of the best exploits that an attacker can take advantage of.

4.3.3 Busybox Crash

BusyBox is known as the Swiss Army Knife of Embedded Linux which is a utility program that groups many common Linux utilities in a tiny executable. It also acts as an efficient alternative to the GNU Coreutils that takes care of the Linux utilities. BusyBox considers the resource constraints and memory constraints in Embedded devices. Although, it does not support all the functionalities that the counterparts offer. BusyBox is also configurable which allows the developers to enable only the utilities that they think are required and opt out of the other utilities. Due to this feature, the end executable is less in size which aids the Embedded devices. The version of BusyBox in Arada Locomate is 1.11 as we can see from *Nmap*.

BusyBox supports commands like chmod, echo, telnet and many more. The hypothesis here is to monitor the reaction of telnet port to an input string of high length. The fuzzer code in Listing 4.1 is the fuzzer program that was used to fuzz the Telnet port. The listing 4.1 contains the Python code snippet for fuzzing the Telnet port 23. The code creates a host variable representing the IP address of the Arada Locomate which is static $IPAddress$: 192.168.0.40, the port variable is declared and initiated to 23 describing the Telnet port number. The size of the string sent is handled using length variable which is used to increase the length of 'C' from 250 until the program crashes. The while loop is used to keep injecting the data into the guest port until an interrupt raises. In the loop, a socket variable is declared, and the socket is defined. Connect () checks if the host and guest port are connected. The output variable will contain a string of a command 'USER' with 'C' and the number of C's is set using length variable. The 'USER' command specifies that the input is for the user name while logging in through telnet. The same is done for the password as indicated in lines 10 and 11. The recv() command helps in getting the reply from the target. If

Listing 4.1 Code snippet of the Port 6666 Fuzzer

```
import sys, socket, time
host = "192.168.0.40" //IP Address
port = int(23) //Port Number
length =256 //Initial length of the string
while (1):
     client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     //Declare a TCP socket
     client.connect((host, port))
     //Connect to user-supplied port and IP address
     output = "USERNAME " + "C" * length
     client.sendall(output.encode('utf-8'))
     output1 = "PASSWORD " + "C" * length
     client.sendall(output1.encode('utf-8'))
     print(client.recv(1024)) //Recieve Reply
     client.close() //Close the connection
     print ("Length Sent: " + str(length)
     length += 100
```

it does not reply or if it hangs, the recv() keeps waiting for the reply which will cause the fuzzer to time out indicating a crash. The sendall() function sends the string to the port. Then, the length of the string is printed in the command line, and the length is increased by 100.

After a certain length, telnet stops accepting data, and the fuzzing program also stops sending data into it representing that the telnet service is crashed causing a denial of service. To recheck the result of the above process, a user can try logging into the system. The telnet does not react for the command and remains frozen. This lead to a successful crash of the Telnet port which can cause problems with hardwares functionality, although this only occurred once. Thus, this attack constitutes a failure. The above attack can further be investigated if there is a presence of buffer overflow or just a crash using GDB debugger.

To regain the telnet functionality, one has to restart the hardware which is not a reliable option when the car is running since it can lead to a dangerous situation for the driver and the other vehicles connected to the network.

4.3.4 USB Daemon

The program that opens the port 6666 can be obtained using a Linux command which gives the program that enables the port to move to open state. The command is as shown below.

$fuser\ \ tcp/6666$

The fuser command is used to identify the file or process using the specified socket. The program that uses the port 6666 is *usbd*. However, the port is only opened for 10 seconds every 1.5 minutes. Also, the port 6666 has a history of incidents where it was reported being used by Trojans and Malware in different systems [11]. This lead to a hypothesis over the *usbd* program. The program is reverse engineered and discussed in Chapter 5.

4.3.5 Admin user vulnerability

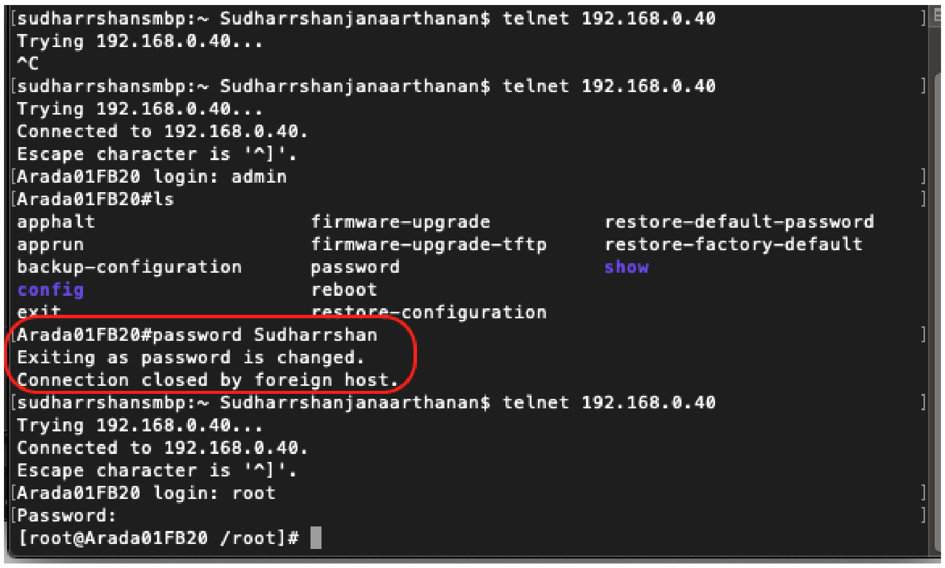
The admin user is for the real world users who have commands and applications that enable the users to run applications they require. The admin user does not need a password and has limited capability. Figure 4.3 shows the applications for the admin user. One of the options is the *password* option. It takes the new password string as an argument as shown below that changes the root password to *Sudharrshan*. The problem here is, it does not check for the current password and directly changes the root password to *Sudharrshan* which is a major flaw with the application. Any user can use this flaw and take control of the system without trying any exploits. This vulnerability is a major success. Figure 4.3 shows the entire process explained above.

$password\ \ Sudharrshan$

4.3.6 Encryption keys and certificates

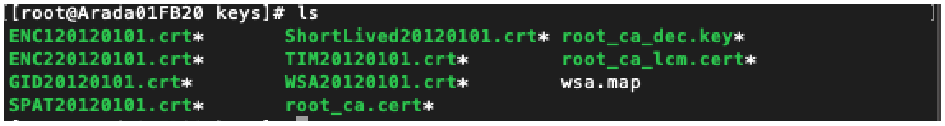
The previous vulnerability gives root user privileges which allow taking complete control of the entire device. The */tmp* folder, in general, will contain the keys created for encryption in the machines. The */tmp* had various keys and certificates as shown in Figure 4.4

Figure 4.5 shows the decoded keys that helps reading the messages sent over to other devices. The keys pave the way for sniffing the data between two devices. Once sniffed, the attacker can



```
[sudharrshansmbp:~ Sudharrshanjanaarthanan$ telnet 192.168.0.40
Trying 192.168.0.40...
^C
[sudharrshansmbp:~ Sudharrshanjanaarthanan$ telnet 192.168.0.40
Trying 192.168.0.40...
Connected to 192.168.0.40.
Escape character is '^]'.
[Arada01FB20 login: admin
[Arada01FB20#ls
apphalt                   firmware-upgrade          restore-default-password
apprun                    firmware-upgrade-tftp     restore-factory-default
backup-configuration      password                  show
config                    reboot
exit                      restore-configuration
[Arada01FB20#password Sudharrshan
Exiting as password is changed.
Connection closed by foreign host.
[sudharrshansmbp:~ Sudharrshanjanaarthanan$ telnet 192.168.0.40
Trying 192.168.0.40...
Connected to 192.168.0.40.
Escape character is '^]'.
[Arada01FB20 login: root
[Password:
 [root@Arada01FB20 /root]#
```

Figure 4.3 Admin user applications



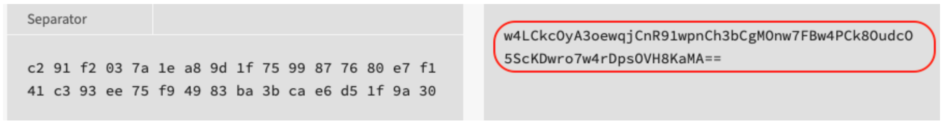
```
[root@Arada01FB20 keys]# ls
ENC120120101.crt*       ShortLived20120101.crt* root_ca_dec.key*
ENC220120101.crt*       TIM20120101.crt*        root_ca_lcm.cert*
GID20120101.crt*        WSA20120101.crt*        wsa.map
SPAT20120101.crt*       root_ca.cert*
```

Figure 4.4 Encryption keys & certificates

spoof the data with malicious content, and jam the communication for a man-in-the-middle attack. This procedure also constitutes a flaw of the device.

4.3.7 Firmware analysis

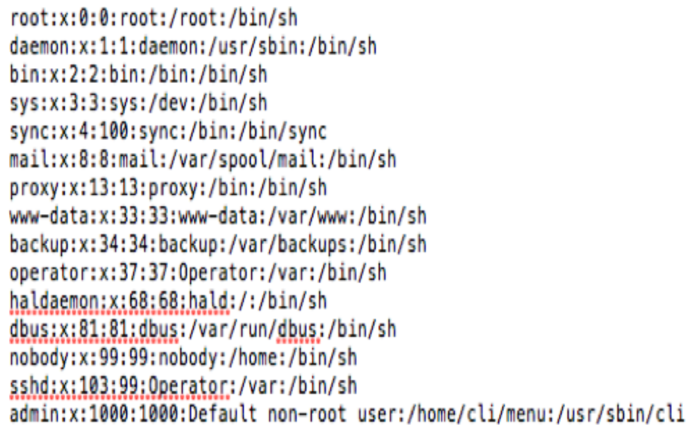
Binwalk is a Kali Linux based tool that extracts the root file system from the compressed files and provide the files in the root file system. Chapter 2 describes the usage of this tool. Important findings are:



```
c2 91 f2 03 7a 1e a8 9d 1f 75 99 87 76 80 e7 f1
41 c3 93 ee 75 f9 49 83 ba 3b ca e6 d5 1f 9a 30
```

```
w4LCkcOyA3oewqjCnR91wpnCh3bCgMOnw7FBw4PCk8OudcO
5ScKDwro7w4rDpsOVH8KaMA==
```

Figure 4.5 Base64 decoded private key



```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
operator:x:37:37:Operator:/var:/bin/sh
haldaemon:x:68:68:hald:/:/bin/sh
dbus:x:81:81:dbus:/var/run/dbus:/bin/sh
nobody:x:99:99:nobody:/home:/bin/sh
sshd:x:103:99:Operator:/var:/bin/sh
admin:x:1000:1000:Default non-root user:/home/cli/menu:/usr/sbin/cli
```

Figure 4.6 List of programs and locations

- The etc. folder had passwd file containing execs for each interface and ports. Also, it has keys for dropbear.

- The /usrs local had a python2.6 folder. But it was empty

- /lib had files including socketCAN and watchdog

- /bin had the busybox executable and links of other Linux command executables

As we expected, Busybox is the telnet client, and Dropbear is the used ssh client. The firmware might support python based codes too. We can also see it has support for CAN bus as SOCKET-CAN is present. Also from the files, we can observe the interfaces, users, and locations of executable of each interface as shown in Figure 4.6. Also, the xauth was not present in Figure 4.6 which proves that there is no X server in the firmware.

4.4 Summary

In this chapter, we discussed the various aspects of penetration testing of Arada Locomate On Board Unit and found the open ports and hypothesized over the vulnerabilities that the services of each port had. Based on the hypothesis, various experiments and exploits were discussed, and a suspicious program called usbd was decided to be analyzed more. Also, a firmware analysis was conducted to bolster the findings through penetration testing.

CHAPTER 5. ANALYSIS OF PORT 6666

This chapter discusses the *usbd* program that accepts messages through port 6666. The reverse engineering tools section discusses the two different tools that were used to reverse engineer the program. The *usbd* program section describes the program in detail including the main function and the other functions tied with the program. The hypothesis section describes where the vulnerability for buffer overflow is suspected. The next section describes the fuzzer used to test the 6666 port and the stack data are studied to inspect the reaction of the code to the values sent. Finally, the result is discussed.

5.1 Introduction

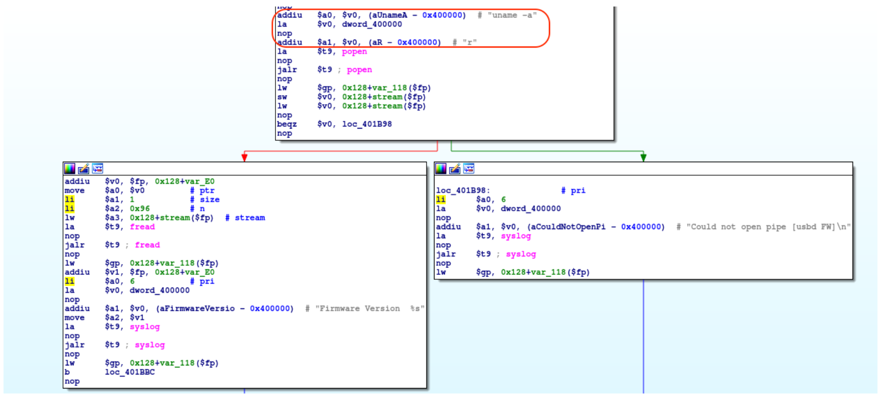
The analysis of the usbd program was motivated due to the fact that port 6666 is mostly used by malware and trojans. IDA pro and Radare2 are the tools that are available for program disassembly. The possibility of a buffer overflow in such ports allows for penetrating through the system. The presence of port 6666 in the program was confirmed by reverse engineering through Radare2, and later stages of the code inspection were performed through IDA Pro. Also, the injection and monitoring of stack were conducted in an emulator with GDB debugger installed as discussed in further sections.

5.2 USBD Program

The code was analyzed using IDA Pro revealing that the executable is a USB Daemon which acts as the setup program that aids the connection between the hardware and the USB drive. In addition to this functionality, this program also logs the Ethernet and WLAN packets in the USB drive. The important functions of this program are described in Table 5.2.

Table 5.1 Switch case and relevant figures

Case	Relevant figure
Case 0,4, default	5.10
Case1	5.6 & 5.7
Case 2	5.8
Case 3	5.9 & 5.10
Case 5	5.11 & 5.12
Case 6	5.6 & 5.7
Case 7	5.9 & 5.10



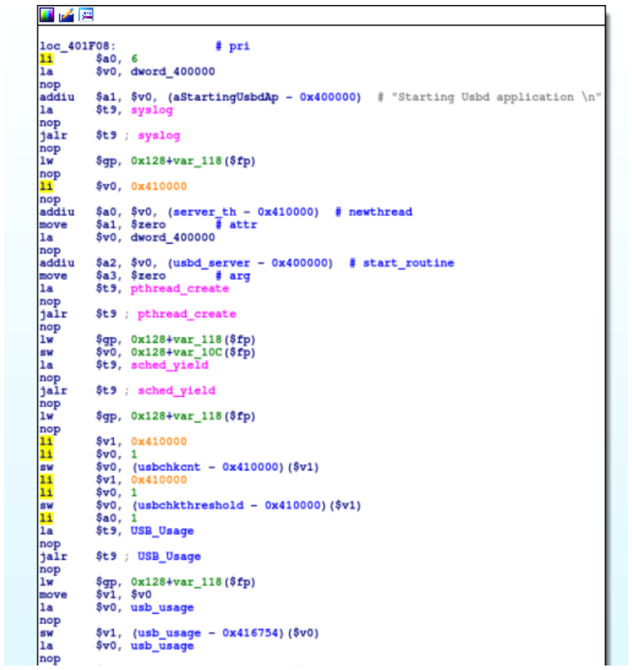
```
addiu   $a0, $v0, (aUnameA - 0x400000)  # "uname -a"
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aR - 0x400000)  # "r"
la      $t9, popen
nop
jalr    $t9 ; popen
nop
lw      $gp, 0x128+var_118($fp)
sw      $v0, 0x128+stream($fp)
lw      $v0, 0x128+stream($fp)
nop
beqz    $v0, loc_401B98
nop
```

```
addiu   $v0, $fp, 0x128+var_E0
move    $a0, $v0         # ptr
li      $a1, 1           # size
li      $a2, 0x96        # n
lw      $a3, 0x128+stream($fp)  # stream
la      $t9, fread
nop
jalr    $t9 ; fread
nop
lw      $gp, 0x128+var_118($fp)
addiu   $v1, $fp, 0x128+var_E0
li      $a0, 6           # pri
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aFirmwareVersio - 0x400000)  # "Firmware Version  %s"
move    $a2, $v1
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x128+var_118($fp)
b       loc_401BBC
nop
```

```
loc_401B98:                 # pri
li      $a0, 6
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aCouldNotOpenPi - 0x400000)  # "Could not open pipe [usbd FW]\n"
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x128+var_118($fp)
```

Figure 5.1 Start of main function

The program is described below and the figures 5.1 to 5.12 show the control flow graph. The *usbd* program acts as a USB daemon used to connect the hardware and the USB device. The main function first checks the firmware details by *uname* command in Linux as shown in Figure 5.1. Then it creates a multithreading application with one thread taking the usage of USB and the other thread going to the usbd_server function as shown in Figure 5.2. This function opens the port 6666 and finishes the network setup functions as discussed in 5.2 and shown in Figures 5.3 & 5.4. After the setup, the read function reads from the file descriptor of the input and then another variable is used to select the case to be executed. The various switch cases in the program are shown in Table 5.3. Figure 5.5 to 5.12 illustrate switch cases 1 to 8 . Once the usbd_server finishes, the control is sent back to the main program and the program terminates.



```
loc_401F08:              # pri
li      $a0, 6
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aStartingUsbdAp - 0x400000)  # "Starting Usbd application \n"
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x128+var_118($fp)
nop
li      $v0, 0x410000
nop
addiu   $a0, $v0, (server_th - 0x410000)  # newthread
move    $a1, $zero       # attr
la      $v0, dword_400000
nop
addiu   $a2, $v0, (usbd_server - 0x400000)  # start_routine
move    $a3, $zero       # arg
la      $t9, pthread_create
nop
jalr    $t9 ; pthread_create
nop
lw      $gp, 0x128+var_118($fp)
sw      $v0, 0x128+var_10C($fp)
la      $t9, sched_yield
nop
jalr    $t9 ; sched_yield
nop
lw      $gp, 0x128+var_118($fp)
nop
li      $v1, 0x410000
li      $v0, 1
sw      $v0, (usbchkcnt - 0x410000)($v1)
li      $v1, 0x410000
li      $v0, 1
sw      $v0, (usbchkthreshold - 0x410000)($v1)
li      $a0, 1
la      $t9, USB_Usage
nop
jalr    $t9 ; USB_Usage
nop
lw      $gp, 0x128+var_118($fp)
move    $v1, $v0
la      $v0, usb_usage
nop
sw      $v1, (usb_usage - 0x416754)($v0)
la      $v0, usb_usage
nop
```

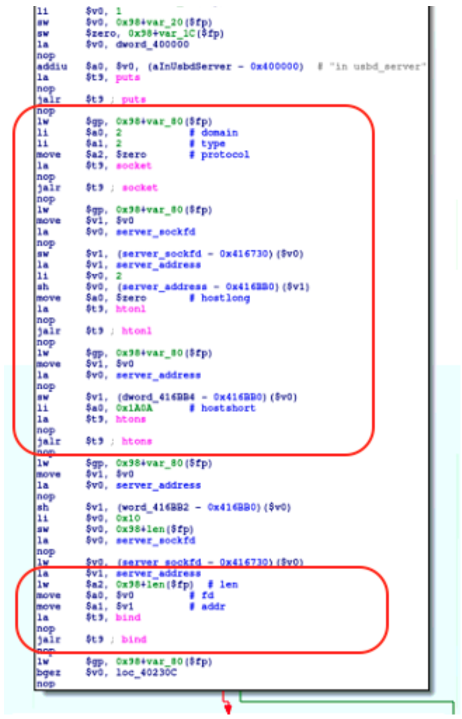
Figure 5.2 usbd_server function call

5.3 Buffer overflow hypothesis

A buffer overflow happens when the size of the input is not moderated either while reading or processing it in the memory. The read() function constitutes the function that reads data from the port. This hypothesis is based on the read function to test if it can acquire more data than specified as a maximum in the function call. If the read() allows reading more than the maximum size specified, then the buffer overflow can be performed which can give access to the program stack. The next section describes the experiments conducted to test the hypothesis.

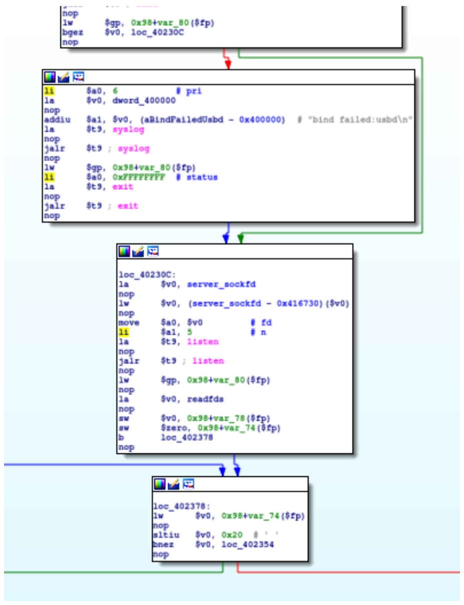
5.4 Injection attack on port 6666

Inspecting the stack of the program requires a debugger running in the system. Arada Locomate is proprietary hardware that has customized Linux version, selected libraries and commands. In addition to it, the file system is a Read-Only file system which restricts the hacker's accessibility. To setup debugger in the hardware, a cross-compiled GDB executable with statically linked libraries



```
li      $v0, 1
sw      $v0, 0x98+var_20($fp)
sw      $zero, 0x98+var_1C($fp)
la      $v0, dword_400000
nop
addiu   $a0, $v0, (aInUsbdServer - 0x400000)  # "in usbd_server"
la      $t9, puts
nop
jalr    $t9 ; puts
nop
lw      $gp, 0x98+var_80($fp)
li      $a0, 2          # domain
li      $a1, 2          # type
move    $a2, $zero      # protocol
la      $t9, socket
nop
jalr    $t9 ; socket
nop
lw      $gp, 0x98+var_80($fp)
move    $v1, $v0
la      $v0, server_sockfd
nop
sw      $v1, (server_sockfd - 0x416730)($v0)
la      $v1, server_address
li      $v0, 2
sh      $v0, (server_address - 0x416BB0)($v1)
move    $a0, $zero      # hostlong
la      $t9, htonl
nop
jalr    $t9 ; htonl
nop
lw      $gp, 0x98+var_80($fp)
move    $v1, $v0
la      $v0, server_address
nop
sw      $v1, (dword_416BB4 - 0x416BB0)($v0)
li      $a0, 0x1A0A     # hostshort
la      $t9, htons
nop
jalr    $t9 ; htons
nop
lw      $gp, 0x98+var_80($fp)
move    $v1, $v0
la      $v0, server_address
nop
sh      $v1, (word_416BB2 - 0x416BB0)($v0)
li      $v0, 0x10
sw      $v0, 0x98+len($fp)
la      $v0, server_sockfd
nop
lw      $v0, (server_sockfd - 0x416730)($v0)
la      $v1, server_address
lw      $a2, 0x98+len($fp)  # len
move    $a0, $v0        # fd
move    $a1, $v1        # addr
la      $t9, bind
nop
jalr    $t9 ; bind
nop
lw      $gp, 0x98+var_80($fp)
bgez    $v0, loc_40230C
nop
```

Figure 5.3 Port 6666 opening

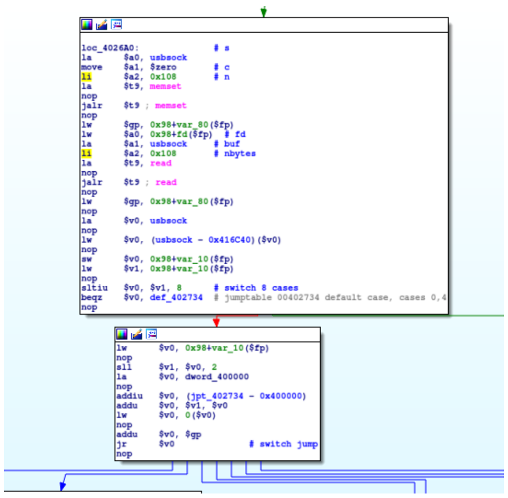


```
nop
lw      $gp, 0x98+var_80($fp)
bgez    $v0, loc_40230C
nop

li      $a0, 6          # pri
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aBindFailedUsbd - 0x400000)  # "bind failed:usbd\n"
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x98+var_80($fp)
li      $a0, 0xFFFFFFFF  # status
la      $t9, exit
nop
jalr    $t9 ; exit
nop

loc_40230C:
la      $v0, server_sockfd
nop
lw      $v0, (server_sockfd - 0x416730)($v0)
nop
move    $a0, $v0        # fd
li      $a1, 5          # n
la      $t9, listen
nop
jalr    $t9 ; listen
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $v0, readfds
nop
sw      $v0, 0x98+var_78($fp)
sw      $zero, 0x98+var_74($fp)
b       loc_402378
nop

loc_402378:
lw      $v0, 0x98+var_74($fp)
nop
sltiu   $v0, 0x20  # ' '
bnez    $v0, loc_402354
nop
```

Figure 5.4 Network setup functions

Listing 5.1 Code snippet of the Port 6666 Fuzzer

```
import sys, socket, time
host = "127.0.0.1" //Address of emulator is through localhost
port = int(6666) // Define the port
length =250 //Initial Length of the string
while (1):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    //Create a socket variable
    client.connect((host, port)) //connect to the port
    output = "C"*length
    client.sendall(output.encode('utf-8'))
    client.close()
    print ("Length Sent: " + str(length)
    length += 100
```



```
loc_4026A0:                 # s
la      $a0, usbsock
move    $a1, $zero          # c
li      $a2, 0x108          # n
la      $t9, memset
nop
jalr    $t9 ; memset
nop
lw      $gp, 0x98+var_80($fp)
lw      $a0, 0x98+fd($fp)   # fd
la      $a1, usbsock        # buf
li      $a2, 0x108          # nbytes
la      $t9, read
nop
jalr    $t9 ; read
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $v0, usbsock
nop
lw      $v0, (usbsock - 0x416C40)($v0)
nop
sw      $v0, 0x98+var_10($fp)
lw      $v1, 0x98+var_10($fp)
nop
sltiu   $v0, $v1, 8         # switch 8 cases
beqz    $v0, def_402734     # jumptable 00402734 default case, cases 0,4
nop
```

```
lw      $v0, 0x98+var_10($fp)
nop
sll     $v1, $v0, 2
la      $v0, dword_400000
nop
addiu   $v0, (jpt_402734 - 0x400000)
addu    $v0, $v1, $v0
lw      $v0, 0($v0)
nop
addu    $v0, $gp
jr      $v0                         # switch jump
nop
```

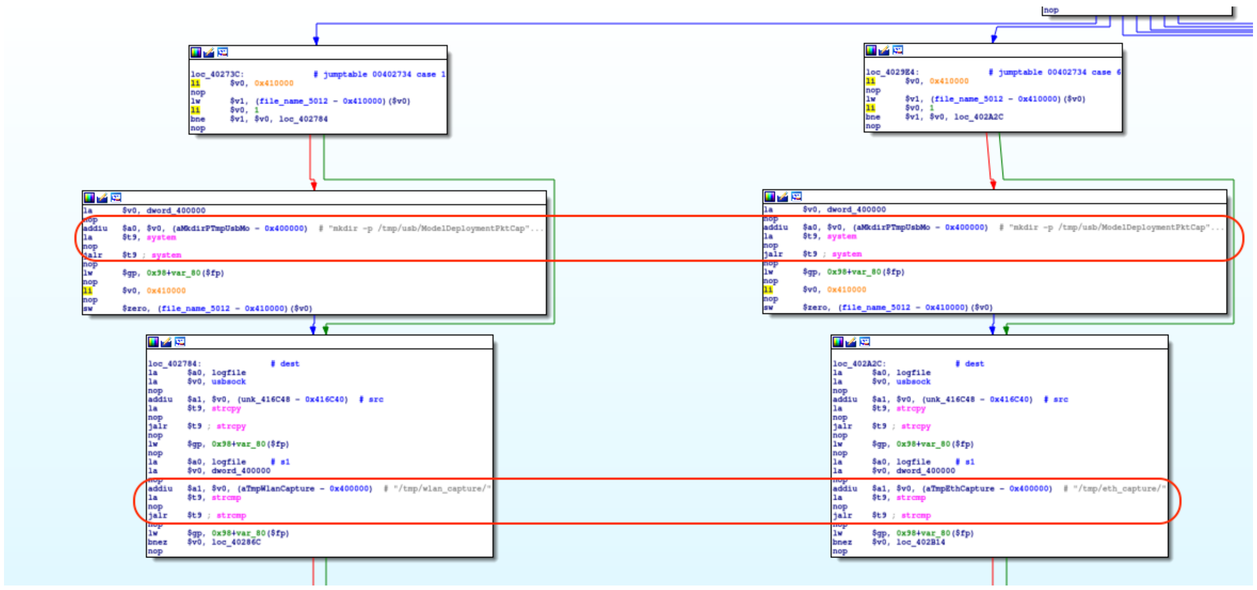
Figure 5.5 Read function

was built that supports MIPS 32 version of the hardware. But, the results show that the GDB was unable to run due to a customized Linux version.

The alternative is to use an emulator that recreates the firmware and root file system of the hardware. An added advantage with the emulator is that it allows the user to perform unlimited control over the firmware. QEMU emulator supports various architecture as discussed, and the setup was done using a MIPS based kernel. GDB packages were installed in the emulator and content of the root file system from firmware analysis was copied to the emulator. After the setup of the emulator, the usbd program is run using GDB which gives full access to the program stack. A fuzzer code as discussed below is run from the host computer. The reaction of the program can be tracked by analyzing the stack data. The following section discusses the fuzzer code, stack data, and the result.

5.4.1 Port 6666 fuzzer

The listing 5.1 contains the Python code snippet for fuzzing the port 6666. The code creates a host variable representing the IP address of the guest, and the port variable is declared and initiated to 6666 describing the port number. The size of the string is handled using length variable which is used to increase the length of 'C' from 250 until the stack overflows. The while loop is used to



```
nop
```

```
loc_40273C:                 # jumptable 00402734 case 1
li      $v0, 0x410000
nop
lw      $v1, (file_name_5012 - 0x410000)($v0)
li      $v0, 1
bne     $v1, $v0, loc_402784
nop
```

```
la      $v0, dword_400000
nop
addiu   $a0, $v0, (aMkdirPTmpUsbMo - 0x400000)  # "mkdir -p /tmp/usb/ModelDeploymentPktCap"...
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
sw      $zero, (file_name_5012 - 0x410000)($v0)
```

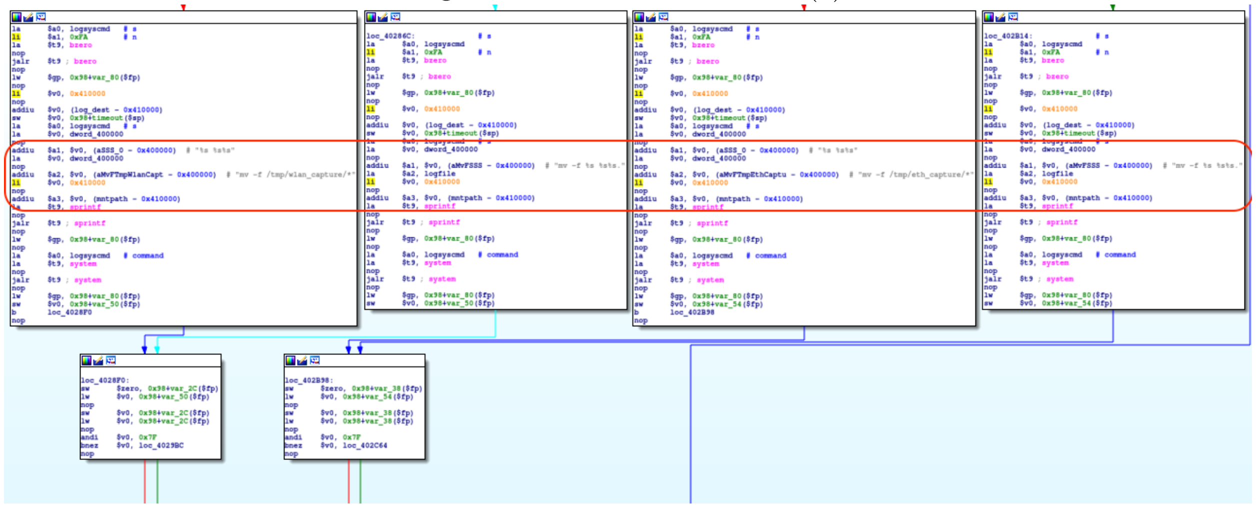
```
loc_402784:                 # dest
la      $a0, logfile
la      $v0, usbsock
nop
addiu   $a1, $v0, (unk_416C48 - 0x416C40)  # src
la      $t9, strcpy
nop
jalr    $t9 ; strcpy
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logfile     # s1
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aTmpWlanCapture - 0x400000)  # "/tmp/wlan_capture/"
la      $t9, strcmp
nop
jalr    $t9 ; strcmp
nop
lw      $gp, 0x98+var_80($fp)
bnez    $v0, loc_40286C
nop
```

```
loc_4029E4:                 # jumptable 00402734 case 6
li      $v0, 0x410000
nop
lw      $v1, (file_name_5012 - 0x410000)($v0)
li      $v0, 1
bne     $v1, $v0, loc_402A2C
nop
```

```
la      $v0, dword_400000
nop
addiu   $a0, $v0, (aMkdirPTmpUsbMo - 0x400000)  # "mkdir -p /tmp/usb/ModelDeploymentPktCap"...
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
sw      $zero, (file_name_5012 - 0x410000)($v0)
```

```
loc_402A2C:                 # dest
la      $a0, logfile
la      $v0, usbsock
nop
addiu   $a1, $v0, (unk_416C48 - 0x416C40)  # src
la      $t9, strcpy
nop
jalr    $t9 ; strcpy
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logfile     # s1
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aTmpEthCapture - 0x400000)  # "/tmp/eth_capture/"
la      $t9, strcmp
nop
jalr    $t9 ; strcmp
nop
lw      $gp, 0x98+var_80($fp)
bnez    $v0, loc_402B14
nop
```

Figure 5.6 Switch case 1 & 6 (a)



```
la      $a0, logsyscmd   # s
li      $a1, 0xFA        # n
la      $t9, bzero
nop
jalr    $t9 ; bzero
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
addiu   $v0, (log_dest - 0x410000)
sw      $v0, 0x98+timeout($sp)
la      $a0, logsyscmd   # s
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aSSS_0 - 0x400000)  # "%s %s%s"
la      $v0, dword_400000
nop
addiu   $a2, $v0, (aMvFTmpWlanCapt - 0x400000)  # "mv -f /tmp/wlan_capture/*"
li      $v0, 0x410000
nop
addiu   $a3, $v0, (mntpath - 0x410000)
la      $t9, sprintf
nop
jalr    $t9 ; sprintf
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # command
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
sw      $v0, 0x98+var_50($fp)
b       loc_4028F0
nop
```

```
loc_40286C:                 # s
la      $a0, logsyscmd
li      $a1, 0xFA        # n
la      $t9, bzero
nop
jalr    $t9 ; bzero
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
addiu   $v0, (log_dest - 0x410000)
sw      $v0, 0x98+timeout($sp)
la      $a0, logsyscmd   # s
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aMvFSSS - 0x400000)  # "mv -f %s %s%s."
la      $a2, logfile
li      $v0, 0x410000
nop
addiu   $a3, $v0, (mntpath - 0x410000)
la      $t9, sprintf
nop
jalr    $t9 ; sprintf
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # command
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
sw      $v0, 0x98+var_50($fp)
```

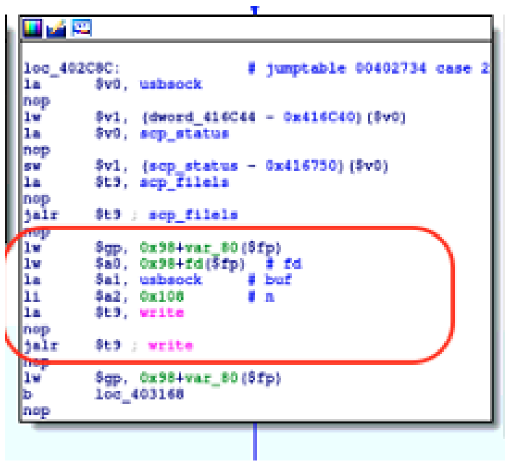
```
la      $a0, logsyscmd   # s
li      $a1, 0xFA        # n
la      $t9, bzero
nop
jalr    $t9 ; bzero
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
addiu   $v0, (log_dest - 0x410000)
sw      $v0, 0x98+timeout($sp)
la      $a0, logsyscmd   # s
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aSSS_0 - 0x400000)  # "%s %s%s"
la      $v0, dword_400000
nop
addiu   $a2, $v0, (aMvFTmpEthCaptu - 0x400000)  # "mv -f /tmp/eth_capture/*"
li      $v0, 0x410000
nop
addiu   $a3, $v0, (mntpath - 0x410000)
la      $t9, sprintf
nop
jalr    $t9 ; sprintf
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # command
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
sw      $v0, 0x98+var_54($fp)
b       loc_402B98
nop
```

```
loc_402B14:                 # s
la      $a0, logsyscmd
li      $a1, 0xFA        # n
la      $t9, bzero
nop
jalr    $t9 ; bzero
nop
lw      $gp, 0x98+var_80($fp)
nop
li      $v0, 0x410000
nop
addiu   $v0, (log_dest - 0x410000)
sw      $v0, 0x98+timeout($sp)
la      $a0, logsyscmd   # s
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aMvFSSS - 0x400000)  # "mv -f %s %s%s."
la      $a2, logfile
li      $v0, 0x410000
nop
addiu   $a3, $v0, (mntpath - 0x410000)
la      $t9, sprintf
nop
jalr    $t9 ; sprintf
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # command
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
sw      $v0, 0x98+var_54($fp)
```

```
loc_4028F0:
sw      $zero, 0x98+var_2C($fp)
lw      $v0, 0x98+var_50($fp)
nop
sw      $v0, 0x98+var_2C($fp)
lw      $v0, 0x98+var_2C($fp)
nop
andi    $v0, 0x7F
bnez    $v0, loc_4029BC
nop
```

```
loc_402B98:
sw      $zero, 0x98+var_38($fp)
lw      $v0, 0x98+var_54($fp)
nop
sw      $v0, 0x98+var_38($fp)
lw      $v0, 0x98+var_38($fp)
nop
andi    $v0, 0x7F
bnez    $v0, loc_402C64
nop
```

Figure 5.7 Switch case 1 & 6 (b)



```
loc_402C8C:                # jumptable 00402734 case 2
la      $v0, usbsock
nop
lw      $v1, (dword_416C44 - 0x416C40)($v0)
la      $v0, scp_status
nop
sw      $v1, (scp_status - 0x416750)($v0)
la      $t9, scp_filels
nop
jalr    $t9 ; scp_filels
nop
lw      $gp, 0x98+var_80($fp)
lw      $a0, 0x98+fd($fp)   # fd
la      $a1, usbsock        # buf
li      $a2, 0x100          # n
la      $t9, write
nop
jalr    $t9 ; write
nop
lw      $gp, 0x98+var_80($fp)
b       loc_403168
nop
```

Figure 5.8 Switch case 2

keep injecting the data into a guest port until an interrupt occurs. In the loop, a socket variable is declared, and the socket is defined. Connect () checks if the host and guest port are connected. The output variable will contain a string of 'C' and the number of C's is set using length variable. The sendall() function sends the string to the port. Then, the length of the string is printed in the command line, and the length is increased by 100.

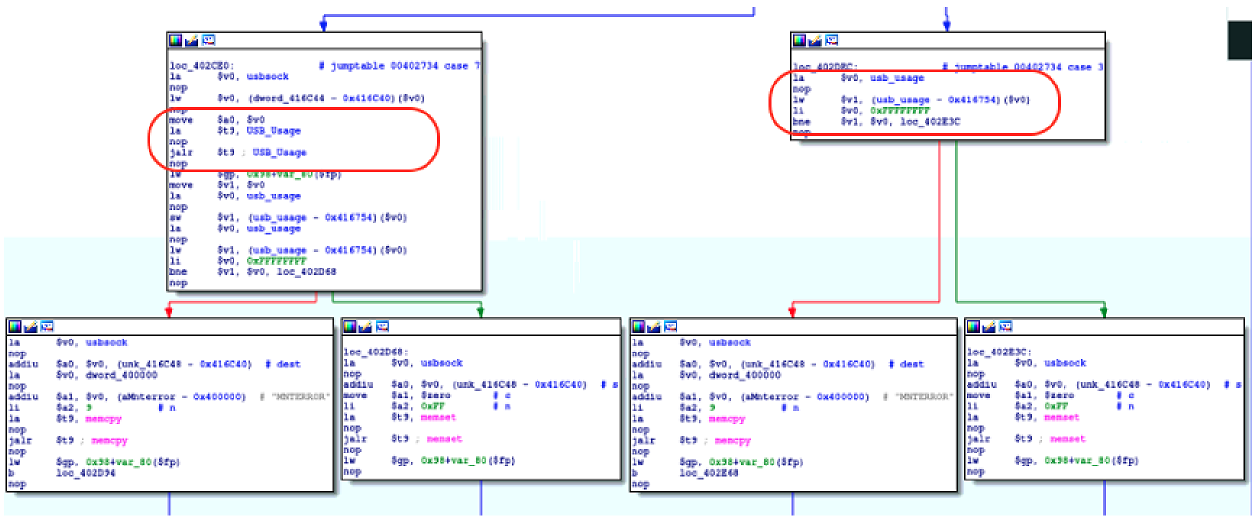
5.4.2 Program stack

The program stack is monitored through GDB debugger which allows us to set breakpoints, observe the buffer until the breakpoint, etc. The program is run in gdb, and the fuzzer program is started from the host computer. The data read from the port is stored in a variable called usbsock with size 256 bytes. Monitor the usbsock in GDB for a possibility of buffer overflow. The command to find the address of usbsock is:

print &usbsock

The above command prints the address location of the start of the variable. In the usbd program, the address of usbsock is 0x416c40. Observe the buffer from this point, and locate the memory locations with 0x43(ASCII value of 'C'). The command to find the memory in general and usbsock in our case is as specified below.

x/FMT Address



```
loc_402CE0:                 # jumptable 00402734 case 7
la      $v0, usbsock
nop
lw      $v0, (dword_416C44 - 0x416C40)($v0)
nop
move    $a0, $v0
la      $t9, USB_Usage
nop
jalr    $t9 ; USB_Usage
nop
lw      $gp, 0x98+var_80($fp)
move    $v1, $v0
la      $v0, usb_usage
nop
sw      $v1, (usb_usage - 0x416754)($v0)
la      $v0, usb_usage
nop
lw      $v1, (usb_usage - 0x416754)($v0)
li      $v0, 0xFFFFFFFF
bne     $v1, $v0, loc_402D68
nop
```

```
loc_402DBC:                 # jumptable 00402734 case 3
la      $v0, usb_usage
nop
lw      $v1, (usb_usage - 0x416754)($v0)
li      $v0, 0xFFFFFFFF
bne     $v1, $v0, loc_402E3C
nop
```

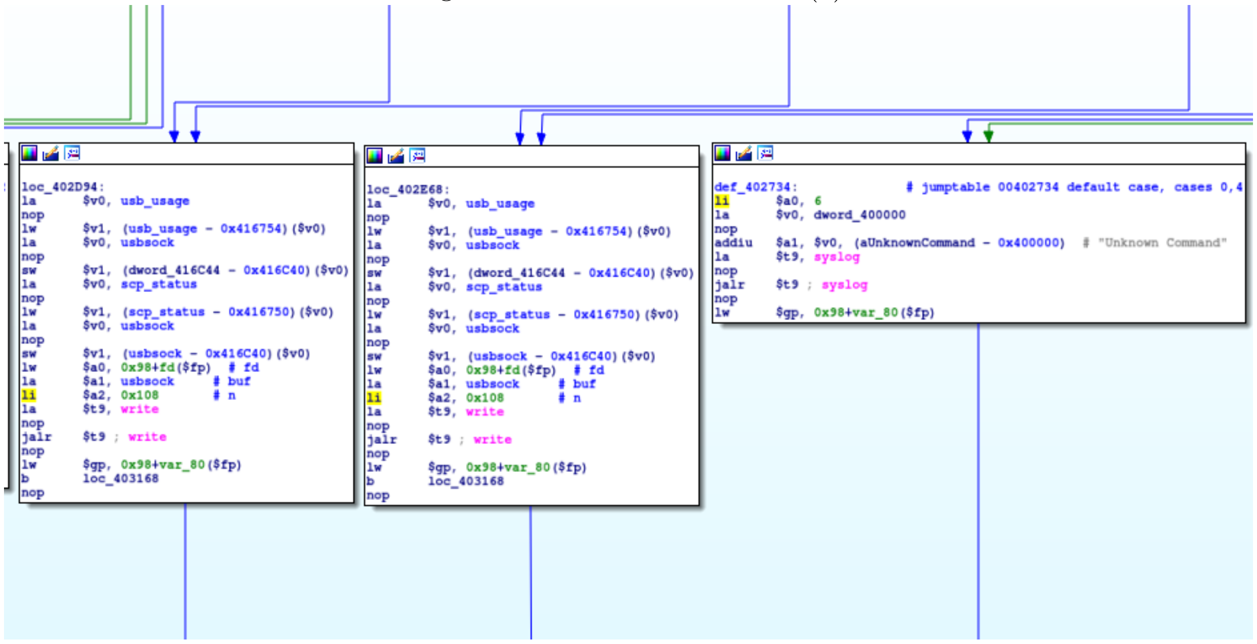
```
la      $v0, usbsock
nop
addiu   $a0, $v0, (unk_416C48 - 0x416C40)  # dest
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aMnterror - 0x400000)  # "MNTERROR"
li      $a2, 9              # n
la      $t9, memcpy
nop
jalr    $t9 ; memcpy
nop
lw      $gp, 0x98+var_80($fp)
b       loc_402D94
nop
```

```
loc_402D68:
la      $v0, usbsock
nop
addiu   $a0, $v0, (unk_416C48 - 0x416C40)  # s
move    $a1, $zero          # c
li      $a2, 0xFF           # n
la      $t9, memset
nop
jalr    $t9 ; memset
nop
lw      $gp, 0x98+var_80($fp)
nop
```

```
la      $v0, usbsock
nop
addiu   $a0, $v0, (unk_416C48 - 0x416C40)  # dest
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aMnterror - 0x400000)  # "MNTERROR"
li      $a2, 9              # n
la      $t9, memcpy
nop
jalr    $t9 ; memcpy
nop
lw      $gp, 0x98+var_80($fp)
b       loc_402E48
nop
```

```
loc_402E3C:
la      $v0, usbsock
nop
addiu   $a0, $v0, (unk_416C48 - 0x416C40)  # s
move    $a1, $zero          # c
li      $a2, 0xFF           # n
la      $t9, memset
nop
jalr    $t9 ; memset
nop
lw      $gp, 0x98+var_80($fp)
nop
```

Figure 5.9 Switch case 3 & 7 (a)

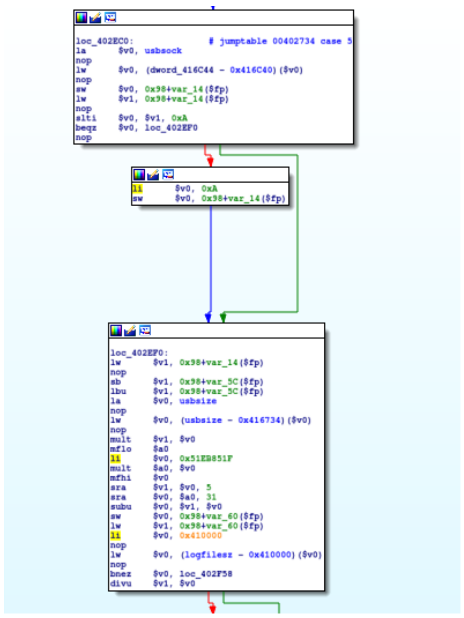


```
loc_402D94:
la      $v0, usb_usage
nop
lw      $v1, (usb_usage - 0x416754)($v0)
la      $v0, usbsock
nop
sw      $v1, (dword_416C44 - 0x416C40)($v0)
la      $v0, scp_status
nop
lw      $v1, (scp_status - 0x416750)($v0)
la      $v0, usbsock
nop
sw      $v1, (usbsock - 0x416C40)($v0)
lw      $a0, 0x98+fd($fp)   # fd
la      $a1, usbsock        # buf
li      $a2, 0x108          # n
la      $t9, write
nop
jalr    $t9 ; write
nop
lw      $gp, 0x98+var_80($fp)
b       loc_403168
nop
```

```
loc_402E68:
la      $v0, usb_usage
nop
lw      $v1, (usb_usage - 0x416754)($v0)
la      $v0, usbsock
nop
sw      $v1, (dword_416C44 - 0x416C40)($v0)
la      $v0, scp_status
nop
lw      $v1, (scp_status - 0x416750)($v0)
la      $v0, usbsock
nop
sw      $v1, (usbsock - 0x416C40)($v0)
lw      $a0, 0x98+fd($fp)   # fd
la      $a1, usbsock        # buf
li      $a2, 0x108          # n
la      $t9, write
nop
jalr    $t9 ; write
nop
lw      $gp, 0x98+var_80($fp)
b       loc_403168
nop
```

```
def_402734:                 # jumptable 00402734 default case, cases 0,4
li      $a0, 6
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aUnknownCommand - 0x400000)  # "Unknown Command"
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x98+var_80($fp)
```

Figure 5.10 Switch case 3 & 7 (b), cases 0, 4 ,default

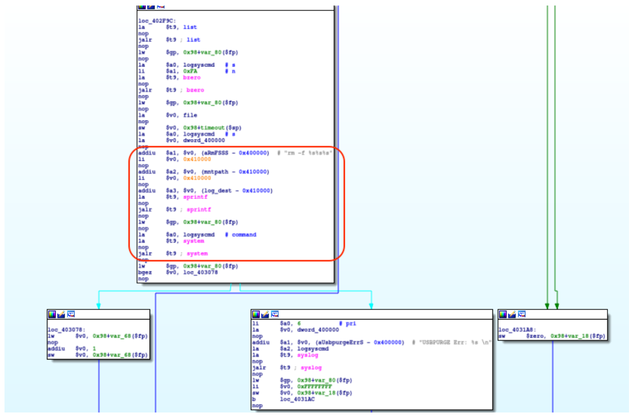


```
loc_402EC0:                  # jumptable 00402734 case 5
la      $v0, usbsock
nop
lw      $v0, (dword_416C44 - 0x416C40)($v0)
nop
sw      $v0, 0x98+var_14($fp)
lw      $v1, 0x98+var_14($fp)
nop
slti    $v0, $v1, 0xA
beqz    $v0, loc_402EF0
nop
```

```
li      $v0, 0xA
sw      $v0, 0x98+var_14($fp)
```

```
loc_402EF0:
lw      $v1, 0x98+var_14($fp)
nop
sb      $v1, 0x98+var_5C($fp)
lbu     $v1, 0x98+var_5C($fp)
la      $v0, usbsize
nop
lw      $v0, (usbsize - 0x416734)($v0)
nop
mult    $v1, $v0
mflo    $a0
li      $v0, 0x51EB851F
mult    $a0, $v0
mfhi    $v0
sra     $v1, $v0, 5
sra     $v0, $a0, 31
subu    $v0, $v1, $v0
sw      $v0, 0x98+var_60($fp)
lw      $v1, 0x98+var_60($fp)
li      $v0, 0x410000
nop
lw      $v0, (logfilesz - 0x410000)($v0)
nop
bnez    $v0, loc_402F58
divu    $v1, $v0
```

Figure 5.11 Switch case 5 (a)



```
loc_402F9C:
la      $t9, list
nop
jalr    $t9 ; list
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # s
li      $a1, 0xFA        # n
la      $t9, bzero
nop
jalr    $t9 ; bzero
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $v0, file
nop
sw      $v0, 0x98+timeout($sp)
la      $a0, logsyscmd   # s
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aRmFSSS - 0x400000)  # "rm -f %s%s%s"
li      $v0, 0x410000
nop
addiu   $a2, $v0, (mntpath - 0x410000)
li      $v0, 0x410000
nop
addiu   $a3, $v0, (log_dest - 0x410000)
la      $t9, sprintf
nop
jalr    $t9 ; sprintf
nop
lw      $gp, 0x98+var_80($fp)
nop
la      $a0, logsyscmd   # command
la      $t9, system
nop
jalr    $t9 ; system
nop
lw      $gp, 0x98+var_80($fp)
bgez    $v0, loc_403078
nop
```

```
loc_403078:
lw      $v0, 0x98+var_68($fp)
nop
addiu   $v0, 1
sw      $v0, 0x98+var_68($fp)
```

```
li      $a0, 6           # pri
la      $v0, dword_400000
nop
addiu   $a1, $v0, (aUsbpurgeErrS - 0x400000)  # "USBPURGE Err: %s \n"
la      $a2, logsyscmd
la      $t9, syslog
nop
jalr    $t9 ; syslog
nop
lw      $gp, 0x98+var_80($fp)
li      $v0, 0xFFFFFFFF
sw      $v0, 0x98+var_18($fp)
b       loc_4031AC
nop
```

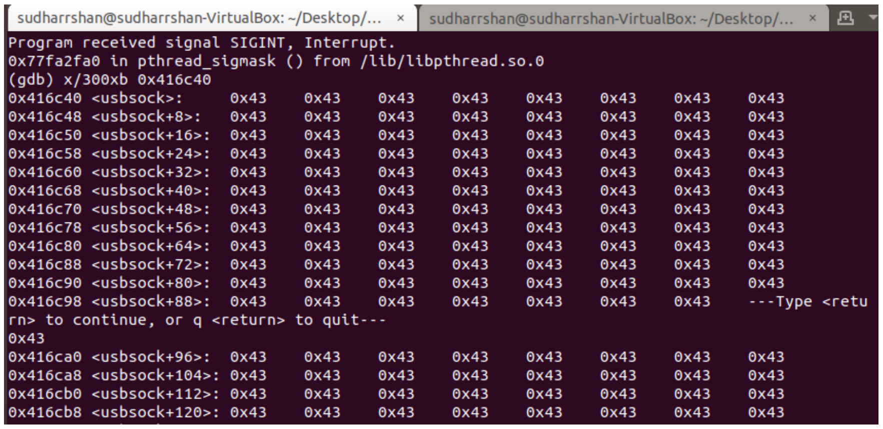
```
loc_4031A8:
sw      $zero, 0x98+var_18($fp)
```

Figure 5.12 Switch case 5 (b)

where x represents the memory and format can be specified to observer bytes or string etc. followed by the address to observe

$x/300xb$ 0x416c40

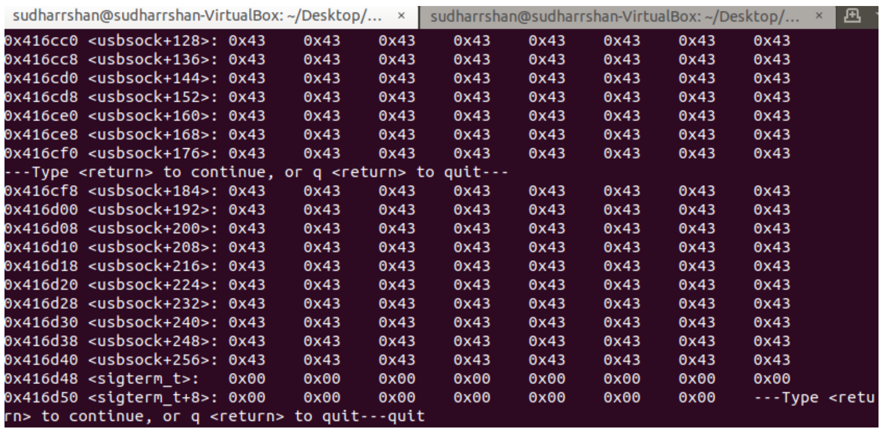
Here the 300 refers to the size of buffer starting from a specified address to display. The 'x' followed by 300, points the command to display the content in hexadecimal and also the 'b' points the command to display byte by byte.



sudharrshan@sudharrshan-VirtualBox: ~/Desktop/...
sudharrshan@sudharrshan-VirtualBox: ~/Desktop/...

```
Program received signal SIGINT, Interrupt.
0x77fa2fa0 in pthread_sigmask () from /lib/libpthread.so.0
(gdb) x/300xb 0x416c40
0x416c40 <usbsock>:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c48 <usbsock+8>:   0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c50 <usbsock+16>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c58 <usbsock+24>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c60 <usbsock+32>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c68 <usbsock+40>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c70 <usbsock+48>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c78 <usbsock+56>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c80 <usbsock+64>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c88 <usbsock+72>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c90 <usbsock+80>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416c98 <usbsock+88>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    ---Type <retu
rn> to continue, or q <return> to quit---
0x43
0x416ca0 <usbsock+96>:  0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416ca8 <usbsock+104>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cb0 <usbsock+112>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cb8 <usbsock+120>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
```

Figure 5.13 Stack content at start of usbsock



sudharrshan@sudharrshan-VirtualBox: ~/Desktop/...
sudharrshan@sudharrshan-VirtualBox: ~/Desktop/...

```
0x416cc0 <usbsock+128>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cc8 <usbsock+136>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cd0 <usbsock+144>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cd8 <usbsock+152>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416ce0 <usbsock+160>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416ce8 <usbsock+168>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416cf0 <usbsock+176>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
---Type <return> to continue, or q <return> to quit---
0x416cf8 <usbsock+184>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d00 <usbsock+192>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d08 <usbsock+200>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d10 <usbsock+208>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d18 <usbsock+216>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d20 <usbsock+224>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d28 <usbsock+232>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d30 <usbsock+240>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d38 <usbsock+248>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d40 <usbsock+256>: 0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x416d48 <sigterm_t>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x416d50 <sigterm_t+8>: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    ---Type <retu
rn> to continue, or q <return> to quit---quit
```

Figure 5.14 Stack content at end of usbsock

The output at the starting of the buffer is as shown in Figure 5.13. The hypothesis over the read() function involves the size of input that the stack can take. The read() specifies the size of the stack as 256 bytes. To test the hypothesis, the content after 256 bytes from 0x416c40 should be observed to see if it exceeds the 256 bytes. Figure 5.14 shows the content of the stack at the end of the 256 bytes.

5.4.3 Result

From Figure 5.14, we can see that the system does not overwrite the buffer with 0x43 after 256 bytes from the start of 0x416c40 also does not crash with a segmentation fault which proves that there is no buffer overflow at the port read() function.

Table 5.2 Description of functions in the usbd program.

Function	Description
Main ()	The main() function initializes the variables required, calls the options() that displays the help manual of the program. It then sets up the multithreading environment. Next, it calls the USB_Usage() function which calculates the free space in the USB and checks if it is under the threshold range. In Parallel, the other thread calls the usbd_server() function, and the further sections describe its functionality. The main program reaches an end after both the thread execution terminates.
USB_Usage()	The USB_Usage() function calls the df Linux command, checks the file system and available memory and returns a flag if there is enough space in the USB.
scp_filels()	The scp_filels() function calls the mv command to move all the files from the source to the destination.
scp_status()	The scp_status() function checks the status of the file copy. It checks if the file copy functionality is completed or not and returns the status as an input to the usbd_server().
bind()	The bind() function is one of the important socket programming functions. It takes the socket file descriptor, socket address, and size of the socket address as input. This function assigns the socket address to the defined socket using the socket file descriptor. It returns 0 on success.
htons()	The htons() function converts the host byte order to network byte order which takes care of the endianness in the network systems.
listen()	The listen() function is also one of the important socket programming functions. It takes the socket file descriptor and backlog as input. This function enables the socket file descriptor to have the ability to accept data. The backlog is the maximum number of pending connections that the port can allow. It returns 0 on success.
accept()	The accept() function is also one of the important socket programming functions. It takes the socket file descriptor, socket address, size of the socket address and flags as input. It accepts the first connection request from listen() and enables the port to accept data.
close()	The close() function is also one of the important socket programming functions. It takes the socket file descriptor as input. Close() frees the file descriptor from the port it is assigned to and terminates the connection.
read()	The read() function is also one of the important socket programming functions. It takes the socket file descriptor, buffer variable, and count as input. This function reads the maximum count of bytes from the file specified by socket file descriptor and stores it in the buffer.
usbd_server()	The usbd_server() function is the most important function of the usbd program. This function starts by printing that the program is in the usbd server and defines the socket as a TCP/IP socket. Port 6666 is declared and htons function is called to take care of the endianness. Then the bind() and listen() functions are called for setting up the port. Next, a thread is created to take care of the interrupts from the program and hardware. The next step is to select the file descriptors to handle. If there is no file descriptor to work on, the control passes back to the thread creation. Once a file descriptor is selected, it calls the accept() to enable the port to accept data. Then, the 264-byte usbsock buffer is declared which is used in the read() to read data. After that, a variable is read and used in a switch case to perform different functions in each case, as described in Table 5.3. Once the switch case terminates, the thread terminates, and the control transfers to main()

Table 5.3 Description of switch cases in usbd_server() program.

Case	Description
Case 0,4, default	This case stores the comment Unknown command in the log file.
Case1	This case creates a folder in the USB called ModelDeployment using the mkdir command. Next, it compares if WLAN capture folder is present and then copies the WLAN packet files from the hardware to the USB. If any error occurs, the log file stores the status.
Case 2	This case calls the scp_status() function to check the file copy status
Case 3	This case is similar to case 7, but it overwrites the data if the USB is full.
Case 5	This case rechecks the USB size, calls the list() which uses the ls command to list all the files in the folder. Next, the files specified in the list() are flushed out from the USB using the rm command to free space.
Case 6	This case creates a folder in the USB called ModelDeployment using the mkdir command. Next, it compares if the ethernet capture folder is present and then copies the ethernet packet files from the hardware to the USB. If any error occurs, the log file stores the status.
Case 7	This case calls the USB_Usage() function to check the free space available in the USB.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

Arada locomate is one of the V2V devices in the market. It has a customized Linux version for the OS. A detailed port analysis of the hardware can reveal any flaws present in the implementation and give the developers a way to alter the programs before deployment. Penetration testing was done to gain an in-depth understanding of the hardware. Port analysis and program analysis combined provided a clear idea of the implementations of software in the device. The hypothesized buffer overflow vulnerability was also tested showing that the program using port 6666 was resistant to buffer overflow attacks. The results of the *vmsplice* experiments also show that the Linux version is susceptible to vmsplice exploit which gives privilege escalation. Generally, sending largely sized inputs on the telnet port is a method to cause a denial of service, but the result of this test showed that the device is safe from this attack. Thus, this study helps to perform required changes in the future firmware upgrades and a secure platform for V2V communications.

6.2 Future work

Few of the tests remain for future work due to lack of time. The flaw hypothesis in chapter 4 discusses a few tests which need to be modified or studied more and customized for the MIPS architecture. The following ideas present the future work based on the possibilities:

- vmsplice() attack: The current exploits are for the i86 architecture. The Arada Locomate OBU is a MIPS architecture based device which uses different registers and commands. The exploit needs to be developed for MIPS architecture and tested

- JTAG: JTAG is a port for hardware debugging. Through JTAG port, firmware can be modified to obtain root privileges on the system. Although, this exploit requires physical access to the device.

- User as attacker: A user knowing this access and details of the device can take advantage over the entire network and can act as an attacker. Future work can study this perspective in depth.

- WPS: WiFi Protected Setup is a standard for secure wireless home networks. This aids in securely connecting other devices to the home network. Usage of a PIN is one of the methods in this protocol. Although this acts as a standard for wireless home networks, there were major vulnerabilities found within the protocol. Brute-force attacks are possible, and it is easier to develop for this protocol. Future work can explore these types of attacks.

BIBLIOGRAPHY

[1] Nmap: Port scanning tool,
https://nmap.org/book/man.html

[2] Metasploit: Exploit database and tools,
https://metasploit.help.rapid7.com/docs

[3] Binwalk: Firmware analysis tool,
https://tools.kali.org/forensics/binwalk

[4] Radare2 reverse engineering framework ,
https://github.com/radare/radare2

[5] IDA Pro disassembler ,
https://www.hex-rays.com/products/ida/

[6] QEMU Emulator: User documentation,
https://qemu.weilnetz.de/doc/qemu-doc.html

[7] QEMU Emulator: Download link,
https://www.qemu.org/download/

[8] GDB Debugger: Download link,
https://ftp.gnu.org/gnu/gdb/

[9] Common Vulnerabilities and Exposures website
https://cve.mitre.org/

[10] vmsplice(): the making of a local root exploit
https://lwn.net/Articles/268783/

[11] History of port 6666

https://www.speedguide.net/port.php?port=6666

[12] Yilin Zhao, *Telematics: safe and fun driving*, *IEEE Intelligent Systems*,vol.17, no. 1, pp. 10-14, Jan. 2002. Available doi: 10.1109/5254.988442

[13] Charlie Miller and Chris Valasek *Adventures in Automotive Networks and Control Units*,2013

[14] Smith, Craig. *The Car Hacker's Handbook: A Guide for the Penetration Tester.*San Francisco, CA, USA: No Starch Press,2016.

[15] Hoppe, Tobias and Kiltz, Stefan and Dittmann, Jana.(2008).Security Threats to Automotive CAN Networks - Practical Examples and Selected Short-Term Countermeasures. In Harrison, Michael D. and Sujan, Mark-Alexander,*Computer Safety, Reliability, and Security(pp. 235-248)*. Berlin, Heidelberg: Springer Berlin Heidelberg

[16] K. Koscher and A. Czeskis and F. Roesner and S. Patel and T. Kohno and S. Checkoway and D. McCoy and B. Kantor and D. Anderson and H. Shacham and S. Savage.(2010).Experimental Security Analysis of a Modern Automobile.*2010 IEEE Symposium on Security and Privacy(pp. 447-462).*

[17] Golson, Jordan, *Car hackers demonstrate wireless attack on Tesla Model S.*(2016). The Verge https://www.theverge.com/2016/9/19/12985120/tesla-model-s-hack-vulnerability-keen-labs

[18] Sahar Mazloom and Mohammad Rezaeirad and Aaron Hunter and Damon McCoy.(2016).A Security Analysis of an In-Vehicle Infotainment and App Platform.*10th USENIX Workshop on Offensive Technologies (WOOT 16).*Austin,TX: USENIX Association.

https://www.usenix.org/conference/woot16/workshop-program/presentation/mazloom,

[19] Checkoway, Stephen and McCoy, Damon and Kantor, Brian and Anderson, Danny and Shacham, Hovav and Savage, Stefan and Koscher, Karl and Czeskis, Alexei and Roesner,

Franziska and Kohno, Tadayoshi.(2011).Comprehensive Experimental Analyses of Automotive Attack Surfaces. *Proceedings of the 20th USENIX Conference on Security(pp. 6–6).*San Francisco, CA: USENIX Association
http://dl.acm.org/citation.cfm?id=2028067.2028073

[20] Linux vmsplice exploit for i86 architecture ,
https://www.exploit-db.com/exploits/5093