Securing ROS

Carl Hildebrandt[1]

***Abstract*— ROS is used by tens of thousands of people in both research and industry. However, ROS was not designed with any security in mind. In this paper, we design a simple robotic system. We then show how an attack can be constructed and implement 3 working attacks. This paper also analyses current security defenses and notes that all require some changes to existing code. This is not ideal as any already implemented ROS code would need to be changed by expert developers to allow for secure operation. We thus design and implement a classifier which can identify attacks 92% of the time. This classifier requires no changes to the existing code base by simply monitoring the communication of the robot.**

I. INTRODUCTION

The robotic operating system (ROS) is an open-source, meta-operating system for any given robot. It provides services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers [5]. ROS was however designed with very little security in mind.

This paper looks at how vulnerable ROS is by designing and implementing both control and attack code from ROS. The design is then implemented on a TurtleBot3 shown in Figure 1. The final implementation is tested and a proposed solution shown.

II. RELATED WORK

The motivation for this work comes from both the robotics community and the software security community. Both communities are very active fields however for this project only the intersection of these fields was considered as related work. A recent study led to the undertaking of this project. The study scanned the internet for unprotected ROS robots [13]. The study did 3 scans, each scan observed over 100 unprotected ROS instances, spanning 28 countries, with over 70% of the observed instances using addresses belonging to various university networks or research institutions.

This study was used as a motivation to find a way to secure the robots without changing the underlying infrastructure effectively. This work was further motivated by the fact that robots are currently not being secured over the internet, meaning that there are many more unsecured robots on private networks. ROS by default runs all communication on port 11311 (TCP) [6]. This TCP connection currently has no security implemented on any of the TCP layers. ROS is currently used by tens of thousands of users [4], leaving the big questions why was security left out of the design?

G.A. vd. Hoorn, a researcher at the DRI and COR at Delft University of Technology and ROS-Industrial contributor

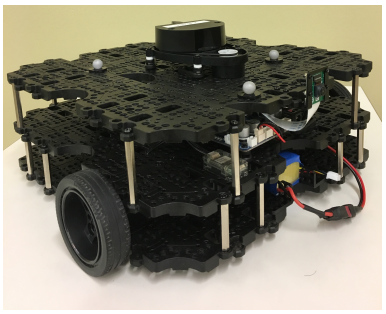

Fig. 1. The TurtleBot3 used for design and implantation of this paper

answered the question. He said that: "security was explicitly (AFAIK, I was not part of the development team, nor am I now) not considered in the ROS 1 design: its goal was to facilitate almost effortless information exchange between (potentially widely distributed) nodes in a computation graph, over a fully trusted network" [2].

This lack of security in the ROS architecture has led to many attempts to secure ROS. ROS 2, the next generation of ROS is currently under development [1]. While ROS 2 has no planned security, there have been talks of adding it, as it tries to address the shortcoming of the original ROS.

Another attempt to add security to ROS is Secure ROS (SROS). SROS is a set of security enhancements for ROS, such as native TLS support for all socket transport within ROS. The use of x.509 certificates permitting chains of trust. Definable name-space globbing for ROS node restrictions and permitted roles. A covenant user-space tooling kit to auto-generate node key pairs, audit ROS networks, and construct/train access control policies[8]. It would seem SROS as successfully achieved what this project set out to do. However as of writing this paper SROS's website is noted as "highly experimental and under heavy development". Thus this solution is still not complete.

There have also been attempts which are less general than ROS 2 or SROS, such as adding message authentication codes to ROS [18]. This work focuses on the use of web tokens for achieving secure authentication of remote clients. More complicated techniques have also been tried such as modifying the source code to allow (D)TLS channels ensuring authentication, authorization, and confidentiality of information exchange between nodes [14].

All these methods, however, share one core attribute they all change how the existing communication is done to achieve security. Changing how communication is handled has numerous drawbacks. Firstly it generally leads to any code developed under the previous communication methods

needing to be factored. Secondly, they secure communication channels generally add overhead to the communication. Extra communication overhead is not always ideal for robots which require real-time information, such as an autonomous ariel vehicles pitch, roll, and yaw controller.

III. DESIGN

During the design phase two major factors where considered. The two design factors were:

- Compatibility
- Low Overhead

Compatibility as a design factor is challenging. We wanted our approach to be easily integrated with existing robot implementations. That means little to no changes to the existing code base. The second design consideration was low overhead. We wanted the final design to require very little computation time so that systems which require real-time information could use our software.

Current implementations such as SROS, message authentication codes and allowing (D)TLS channels all require some changes to the original code base. We thus designed our code to be a completely separate node. That is to say: our node would connect to the already available topics and monitor them. It would then give an output telling the developer if their system was being an attacked or not. What the developers of the robot did with this new information was outside the scope of this project. This design choice can be seen in Figure 2.

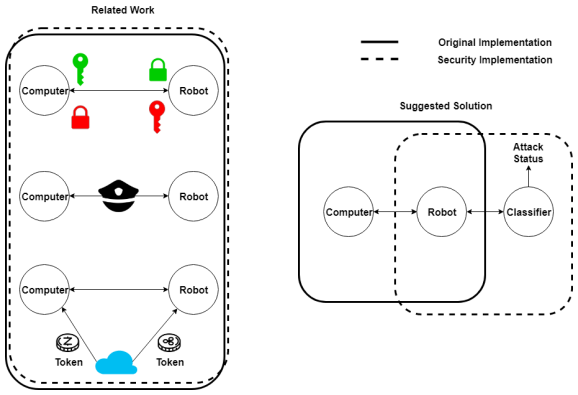

Fig. 2. Current implementation versus our implementation

A system which has very little overhead is a much harder design factor. Low overhead is challenging to design for because there are many factors which contribute towards it such as implementation, hardware, system integration, communication channels used, etc. The implementation details which reduced the overhead are discussed in more detail in Section III-A.3

A. Overview

The first design choice was which robotic platform should be used to implement different prototypes. We chose to implement this technique on a TurtleBot 3 Waffle Pi [9]. We chose this platform as it has open source code, multiple onboard sensors and is widely used in both simulation and practice in the robotics community.

Now that the robotic platform was selected we could focus on the code implementation. We would need three components. We would need control code. This code is what decided the robots mission and actions. We would need attack code. This code attacked the robot and tried to stop it from effectively completing its mission. The final piece of code was the classification code. This would monitor the robot and try and detect whether or not the robot was under attacked. The final design can be seen as the suggested solution in Figure 2.

1) Control Code: The control code was implemented in python using rospy (the python ROS library). The robot was designed to drive at a constant speed. It used a PID controller which tried to minimize the error between the robots current heading and goal direction. The robot was also tasked with collecting 360-degree laser scan data using its onboard lidar sensor. The robots control code is summarized in Algorithm 1:

Algorithm 1: Control Code Design

```
while True do
    Calculate Current Heading
    Calculate Goal Heading
    Record sensor data
    angular velocity = PID(current heading - goal
     heading)
    linear velocity = 1
    Publish Command ( linear velocity angular velocity)
    if Distance To Goal ≤ 0.5 then
        Generate New Random Goal;
    end
end
```

2) Attack Code: To successfully attack the system, we first needed to understand how communication is currently done in ROS. All communication in ROS is done through a ROS master. Any node wanting to publish data advertise this to the ROS master. This can be seen by the original Hokuyo node and the Attack node in Figure 3. Any node then wanting to receive these messages will subscribe to that message type by sending a subscribe request to the ROS master. The ROS master then replies with the publishing nodes network address. The subscribing node then initiates a TCP connection with the publishing node. Once that TCP connection has been negotiated, communication is established.

Thus for an attack to occur, one needs to advertise the messages of the same topic name to the ROS master. The ROS master will then reply with the network address of both the original node and the attack node to any node wanting to subscribe to that topic. The attack node then needs to construct messages with the exact same structure as that of the original node. However, the data fields can be changed to the attackers liking. This entire process can be seen in Figure 3.

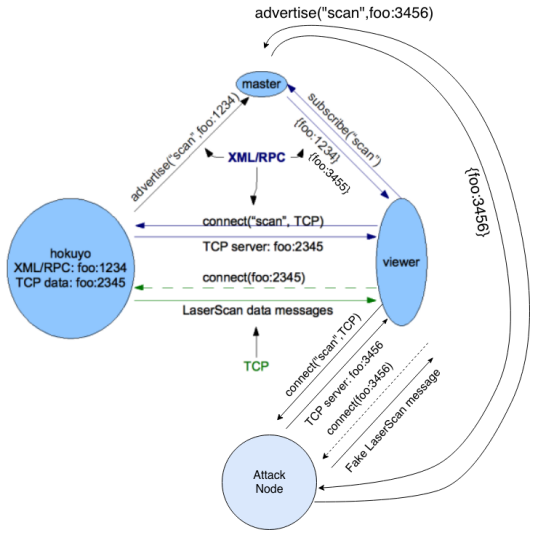

Fig. 3. A figure depicting how the attacks were achieved. Image modified from [7]

Now that an attack could be achieved, we wanted to capture as many types of attacks as possible in this time frame. Many attacks would allow us to show that our approach could be applied to a more general case where the attack was not known to start. The attack code focused on three types derived from[14]:

- Unauthorized Publishing
- Unauthorized Data Access and Manipulation
- Denial of Service (DOS) Attacks

These attacks were able to run entirely independently from the control code. This meant that the attack code was able to run on a separate computer. The only assumption was that the attack computer has access to the same network as that of the control computer (through a physical ethernet or remote wireless connection). This assumption was assumed to be reasonable considering the study in Section II which found over 100 ROS enabled robots accessible through the web [13].

The unauthorized publishing attack published messages which were constructed as if they came from the original publishing node. The attack focused on attacking the location node. The location node known as "vicon_bridge" is a node which takes motion tracking information from a Vicon system [11], and converts it to ROS messages at $200Hz$.

The unauthorized data access and manipulation attack would read the laser scanner information and change it to random noise. This meant that all information collected by the robot was useless. It also meant that any information read by the robot could be accessed and displayed in the attack node.

The final attack was the DOS attack. This attack published command messages to the robot faster than the robot could accept real command messages. The command messages contained stop commands. That meant that the robot would be forced to stop mid-mission. The high rate of attack command messages would also cause the message queue on the robots command topic to overflow, causing the real command packets to be dropped.

3) Classification Code: The final stage was designing the classification code. The first thing to note when designing any classification tool is the type of data it will be classifying. Communication data can be modeled as a sequence of messages. Recurrent neural networks (RNN's) are the state of the art networks for sequential data [16]. They also have the added benefit for allowing sequences of an unknown length to be classified[15]. Both these attributes meant that an RNN is ideal for this application.

RNN's have a known deficiency when it comes to large amounts of data known as the long-term dependencies problem [12]. Robotics applications are sometimes implemented with high-frequency communication. For instance, the position node in our implementation posted messages at $200Hz$. That means if a basic RNN were used, it would need at least 200 inputs to capture a single second worth of data. If the attack started a few seconds into the run, the trends in the first second of robot behavior would have been forgotten.

It was thus decided to use a long short term memory (LSTM) network. LSTM networks solve the long-term dependencies problem by having a long and short term memory unit built into them. This allows LSTM's to maintain a previous state while also taking the current state into account. An example of an LSTM cell is shown below in Figure 4

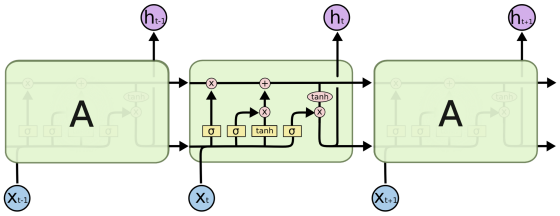

Fig. 4. A standard design for LSTM cells [10]

Now that the long-term dependency problem had been solved, we needed to take into consideration the design constraints described in Section III. The final design constraint was to have a low overhead. To achieve this a very simple network was designed. The network contained a single layer of 160 LSTM cells, followed by another layer containing a single neuron. The network consisted of a total of $310,951$ trainable parameters. $310,951$ trainable parameters are nothing compared to many state of the art networks with consist of many millions of parameters [17]. This was all done with the intent to reduce the complexity and thus reduce the overhead.

IV. RESULTS

The results are broken up into two phases. Phase 1 are the results from the control and attack of the robot. Phase 2 are the results from the classification code.

A. Phase 1

Firstly the control code was tested. The robot was tested with three objectives in mind:

- Did the robot reach its goal?
- Did the robot record data using the laser scanner?
- Did the robot generate a new position once it reached the goal?

To confirm if these objectives were completed, the robots position, goal, and scanner data were printed to screen. An example of the recorded data is shown below in Figure 5.

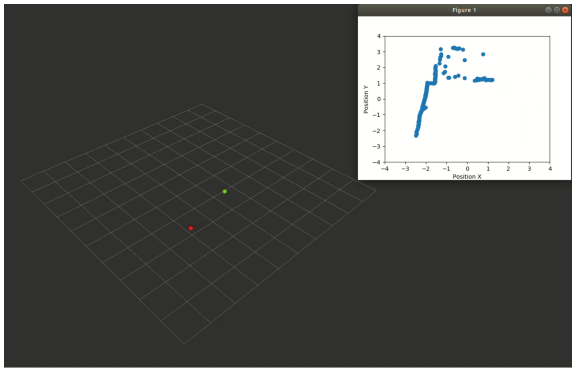

Fig. 5. An example of the outputted data. The red dot is the robots position, the green dot the robots goal and the blue dots are laser scan readings.

Note that in Figure 5, the noise in the laser scan data is due to netting next to the robot. Once the robot passed all evaluations, the robots attack code was tested. The attack code was evaluated with three objectives in mind:

- Could the position of the robot be compromised?
- Could the robots laser scan be read and compromised?
- Could the robot's motors be stopped using a DOS attack?

Testing was done by running the robots control code and launching each of the attacks from a separate computer running the attack code. When the position of the robot was attacked it was noted that the red dot, which notes the robot's position, teleported around all over the map. This caused the robot to jerk as it drove. The jerking was due to the control codes PIDs controllers trying to compensate for the positional noise.

The second attack was done by first reading and then overriding the scanner data. The scanner data was completely over ridding hiding any detected obstacles such as walls. This could be dangerous for the robot or operators as the robot was in essence completely blind. An example of the robots scanner data during the attack is shown below in Figure 6.

The final attack was monitored by looking at the robots current velocity. It was hoped that by flooding the robots command messages with stop commands, the robot would come to a halt or reduce its velocity. The attack was successful causing the robot to became almost completely stationary during that attack.

B. Phase 2

The evaluation of the classifier was split into two sections, training and testing. This is standard in machine learning, however here it had another purpose. I was unsure if this approach was even feasible. Thus I did an initial evaluation

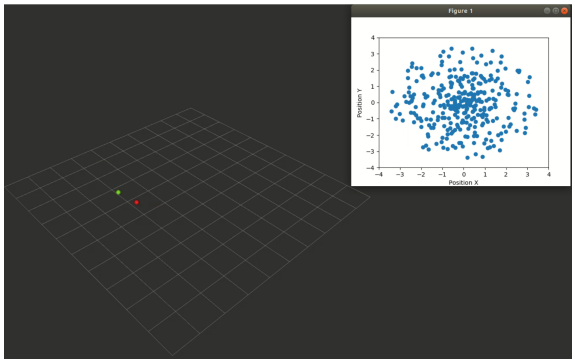

Fig. 6. An example of the robots scan data being attacked. Note that no obstacles such as walls can be seen in the data.

of the training phase to determine if the approach could successfully identify attacks. If the training resulted in a very low accuracy, it could be assumed that the network could not detect attacks and thus the approach would need to be changed. On the other hand, the testing phase not only showed how well the network did on unseen data but also gave us insight into how well the network would work in a real implementation.

To do both training and testing, first data was needed from phase 1. Recording the data was done using rosbags. Bags are the primary mechanism in ROS for data logging[3]. Bags allow data to be recorded and played back at the same rate and order as they were received. A total of 206 bags were recorded each 20 seconds long. 103 bags contained no attack, 34 contained an unauthorized publishing attack, 34 contained a data manipulation attack and 35 contained DOS attacks. Altogether roughly 70 minutes of live communication was recorded.

The data was then broken up into 80% training data and 20% testing data.

1) Training: During training, data was fed one entire bag at a time. That meant that the classifier was given all 20 seconds of recorded data. The classifier would then give a single classification out at the end of the 20 seconds. This was done as a proof of concept for the design.

A better approach would have been to predict a classification for each message individually. Changing to this implementation was easy, and would only require a single flag to be set. Recording data where each message was given a label was more difficult. It could be achieved using two approaches. Either each message to contain a label, meaning that each of the standard messages in ROS would need to modified to contain a label field. Alternatively, another approach would be to send out another message containing the label each time some communication occurred. This would require a complicated implementation to allow for seamless message synchronization. Due to the attack and control codes already being developed for the scope of this project and considering the time frame, this was left as future work. The results however are a proof of concept that this design works.

The best model was able to achieve an accuracy of 100%.

The model's accuracy is shown in Figure 7 below. This was surprising, as it meant the model was very clearly able to differentiate between communication containing attack messages and no attack messages. Note that these messages contained no information about the sender. They also did not contain any information other than the command or data of the message. That means the classifier was able to learn the trends of the robot and determine when a message did not follow this trend.

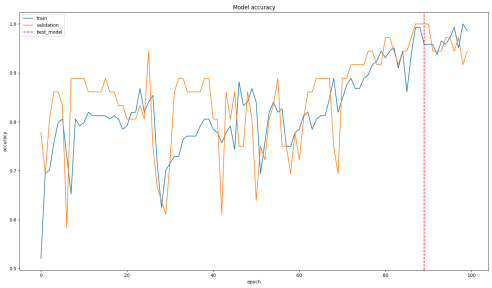

Fig. 7. The accuracy of the RNN LSTM network. The vertical red line indicates the selected network.

2) Testing: The final evaluation of the LSTM network was done by running unseen test data through the classifier. The network was able to identify attack messages with a 92% accuracy. This was very interesting as it meant that using this technique 92% of messages could be dropped by the robot before they ever reached it. That is a huge security improvement over the current nonsecure robots.

However, for this to be feasible the network needed to have very little overhead. During the testing phase message was classified in $53.40\mu S$. The fastest data stream was $200Hz$. That means that a message on this topic would be received every $5000\mu S$. Thus the classification could be done roughly 100 times faster than messages could be sent. It was thus deemed that the overhead of the classifier was sufficient for the proposed design.

V. CONCLUSION

This paper constructs and then demonstrates how attacks to ROS systems are performed. The paper is able to attack the ROS system with a computer which has no physical attachment to the robot other than a connection to the same network. The attacks are all very successful and work 100% of the time.

This paper presents a proposed solution for the current ROS security problems. The proposed solution is a classifier which can learn correct robot behaviors and identify messages which fall outside this range of behaviors. This allows the classifier to find adversarial messages trying to alter the current behavior of the robot. This paper presents an approach using an RNN with LSTM cells. The network was chosen for its very small overhead of just $53.40\mu S$. This overhead is 100 times smaller duration than the faster message being sent. That means this implementation could be used for robots with even the most critical real-time communication needs.

The implementation can also be appended to any system without modification to the existing code. This is a huge benefit over other proposed solutions all of which would require some of the existing code to be refactored or changed. For future work an implementation which feeds back whether the system is being attacked or not can be tested. This would allow the adversarial communication to be dropped before it ever reaches the robot.

APPENDIX

All code can be found in my git repository at the URL:

`https://github.com/hildebrandt-carl/SoftwareSecurityFinalProject`

REFERENCES

[1] ROS 2. `https://design.ros2.org/`. Accessed: 2018-12-06.

[2] ROS answers. `https://answers.ros.org/question/247267/is-ros-20-secure-really-more-secure-than-ros-1/`. Accessed: 2018-12-06.

[3] ROS bags. `https://wiki.ros.org/Bags`. Accessed: 2018-12-07.

[4] ROS history. `http://www.ros.org/history/`. Accessed: 2018-12-06.

[5] ROS introduction. `https://wiki.ros.org/ROS/Introduction`. Accessed: 2018-12-09.

[6] ROS technical overview. `https://wiki.ros.org/ROS/TechnicalOverview`. Accessed: 2018-12-06.

[7] ROS technical overview. `http://wiki.ros.org/ROS/Technical%20Overview`. Accessed: 2018-12-09.

[8] SROS - secure ros. `http://wiki.ros.org/SROS`. Accessed: 2018-12-06.

[9] TurtleBot 3 waffle pi. `http://www.robotis.us/turtlebot-3-waffle-pi/`. Accessed: 2018-12-06.

[10] Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2018-12-07.

[11] Vicon motion capture system. `https://www.vicon.com/`. Accessed: 2018-12-06.

[12] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. The problem of learning long-term dependencies in recurrent networks. In *Neural Networks, 1993., IEEE International Conference on*, pages 1183–1188. IEEE, 1993.

[13] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. *arXiv preprint arXiv:1808.03322*, 2018.

[14] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass, and Peter Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98:192–203, 2017.

[15] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[16] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[17] Suraj Srinivas and R Venkatesh Babu. Learning the architecture of deep neural networks. *CoRR abs/1511.05497*, 84, 2015.

[18] Russell Toris, Craig Shue, and Sonia Chernova. Message authentication codes for secure remote non-native client connections to ros enabled robots. In *Technologies for Practical Robot Applications (TePRA), 2014 IEEE International Conference on*, pages 1–6. IEEE, 2014.