

Advanced Network Security Lab

Session 3: Attacks to the Routing Protocol for Low-Power and Lossy Networks (RPL)

Dominik Roy George
d.r.george@tue.nl

Savio Sciancalepore
s.sciancalepore@tue.nl

March 06, 2024

1 Introduction

The 3rd Lab Session provides you with an example for the final Lab-Assignment, based on an emulated attack to the protocol Routing Protocol for Low-power and lossy networks (RPL). In this document, we show what the *Design* of the attack and the defense should look like, the structure and exemplary content of the report for the final assignment, and how to report the results. Based on the gained experience from the previous lab sessions, you should be able to focus on reference Operative System (OS), namely, *contiki-ng*, to design a concrete attack to RPL. This document includes two main sections: (i) Attack Design and (ii) Defence Design.

2 Attack Design

In the following, we present the design of the *Rank Attack* to the RPL protocol. Specifically, we focus on the way of carrying out the attack through the manipulation of the rank of the adversarial node based on the observation of the ranks of legitimate nodes in the network. First, we show how to design the attack, with reference to the number of devices necessary to realize the attack and the software necessary to verify that the attack is working. Next, we show how to force the topology at the MAC-layer to adhere to a given plan. Last, we describe how to practically deploy the attack on the devices.

2.1 Hardware and Software Planning

For the *Rank attack*, it is necessary to have (at least) 5 nodes (in our case, *Sensortags CC2650* devices). One node takes the role of the border router, which is the root node of the network and routes packets to/from the Internet of Things (IoT) network. Moreover, we need (at least) two nodes to take the role

of *simple nodes*, connecting wirelessly to the border router. To sniff the traffic exchanged between the nodes, we need (at least) one sniffer node, in charge of capturing the traffic and analyzing it through *Wireshark*. In addition, to carry out the attack, we need (at least) one node to behave maliciously. Therefore, we need in total (at least) five devices to realize the attack.

In Fig. 1, we visualize two topologies: the left-most figure shows the topology and traffic flow in the regular (benign) scenario. In contrast, the right-most figure shows the intended (malicious) topology and traffic flow after the successful execution of the attack, where the traffic flow is diverted toward the malicious node.

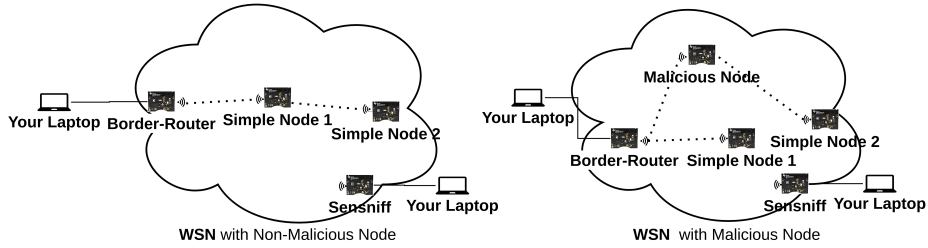


Figure 1: Regular (benign) topology and traffic flow (left), and modified (malicious) ones (right).

2.2 Forcing the Topology

As shown in Fig. 1, the *simple node 2* should communicate with *simple node 1*. However, as devices will be deployed very close to each other, *simple node 2* will likely see the *border-router* as a node directly connected to it, thus selecting this latter as its *preferred parent*. So, it will not connect to *simple node 1* and not to any potentially-malicious node, making it impossible to reproduce the attack we want to deploy.

To carry out the attack as planned, we need to force the *simple node 2* to pass through another node to reach the *border-router*. In summary, we need to *force the topology*, by explicitly letting the *simple node 2* to drop all the packets received directly (at the MAC-layer) from the *border router*.

To force a node to drop packets from a certain node, it is necessary to know the *MAC-addresses* of the two nodes involved.

To acquire such information, it is necessary to read the *MAC-address* from the nodes. In particular, we first need the *MAC-address* of the border router. This can be acquired by issuing the command *tunslip*, as shown in Lab Session 1, Fig. 2. In the Figure, we are able to read the highlighted *MAC-address* in hex format: `fd00::212:4b00:c4a:6c87`. In the figure, we also see the IP addresses of the nodes which are not flashed with the border router stack. As we defined the *MODULES += os/services/shell* line in the *Makefile* of the *hello-world* program (see Lab Session 1), we can log in into these devices and, after pressing

```

dong@dongrgeorge:~/Contiki-ng$ sudo ./tools/serial-to/tunslip6 -s /dev/ttyACM0 fd00::1/64
*****SLIP started on "/dev/ttyACM0"
opened tun device "/dev/tun0"
ifconfig tun0 inet 'hostname' mtu 1500 up
sh: line 1: hostname: command not found
ifconfig tun0 add fd00::1/64
ifconfig tun0 add fe80::0:0:0:1/64
ifconfig tun0
tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST> mtu 1500
    inet6 fe80::1: prefixlen 64 scopeid 0x20<link>
    inet6 fe80::1da0:e62e:e487:6d5c: prefixlen 64 scopeid 0x20<link>
    inet6 fd00::1: prefixlen 64 scopeid 0x0<global>
    unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
    RX packets 0  bytes 0  (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0  (0.0 B)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

[INFO: BR ] Waiting for prefix
*** Address: fd00::1 => fd00::0000:0000:0000
[INFO: BR ] Waiting for prefix
[INFO: BR ] Server IPv6 addresses:
[INFO: BR ] fd00::212:4b00:c4a:6c87
[INFO: BR ] fe80::212:4b00:c4a:6c87

```

Figure 2: Reading the MAC address of the Border Router.

enter multiple times, we can access the interactive shell of *contiki-ng*, Fig. 3 shows how to read the *MAC-address* of the *simple nodes*.

```

dong@dongrgeorge:~/Contiki-ng$ sudo ./tools/serial-to/serialdmm -b115200 /dev/ttyACM0
[sudo] password for dong:
connecting to /dev/ttyACM0 [OK]
#0012,4b00,0c4a,6c87>
#0012,4b00,0c4a,6c87>
#0012,4b00,0c4a,6c87>
#0012,4b00,0c4a,6c87> Hello, world
#0012,4b00,0c4a,6c87> mac-addr
Node MAC address: 0012,4b00,0c4a,6c87
#0012,4b00,0c4a,6c87>

```

Figure 3: Reading the MAC address of the Simple Nodes.

Thus, we now know the *MAC-Addresses* of the border router and the specific simple node.

As a next step, we need to copy the entire *Contiki-ng* folder to a new folder, containing all the sub-folders and files. Hence, we need an individual *contiki-ng* folder for every change to be taken in the OS. Otherwise, the changes taken for specific nodes such as for the *simple node 2* or the *malicious node* would affect also the behavior of other nodes.

Then, we need to force the topology at the *MAC-layer*, using the MAC addresses previously acquired.

In Contiki-ng, the *MAC-Layer* is located in the folder *contiki-ng/os/net/mac/*, and specifically, the code for processing the incoming packets is located in the folder *contiki-ng/os/net/mac/framer*. To force the topology, we need to edit the file *framer-802154.c*, and specifically, the function *parse(void)*, which contains the code for extracting information from the received packets. At line 239 of the file *framer-802154.c*, we need to define the *MAC-address* of the border-router in decimal format:

Listing 1: framer-802154.c.

```

const linkaddr_t linkaddr_br = { { 0, 18, 75, 0, 12, 101, 190, 7
} };

```

In addition, at lines 289-291, we need to add the following code, to check if the source address of the node matches with the address defined in the file and, in

case yes, to discard the packets.

Listing 2: framer-802154.c.

```
if(linkaddr_cmp(&linkaddr_br,(linkaddr_t *)&frame.src_addr)){  
    return FRAMER_FAILED;  
}
```

If the condition results *true*, then the packet is dropped; otherwise, the packet is propagated to the next function call. The change in the code applies only for *Simple Node 2*, while for the remaining nodes we use the unchanged *framer-802154.c* file.

To verify the correct execution of the code, it is necessary to plug in: (i) the border router; (ii) the sniffer, (iii) the *Simple Node 1* and, at last, (iv) the *Simple Node 2*. Fig. 4 shows that the traffic originating from the node whose MAC address ends with *6c87* is routed through the node with the MAC address ending with *4906*, and not through the border-router.

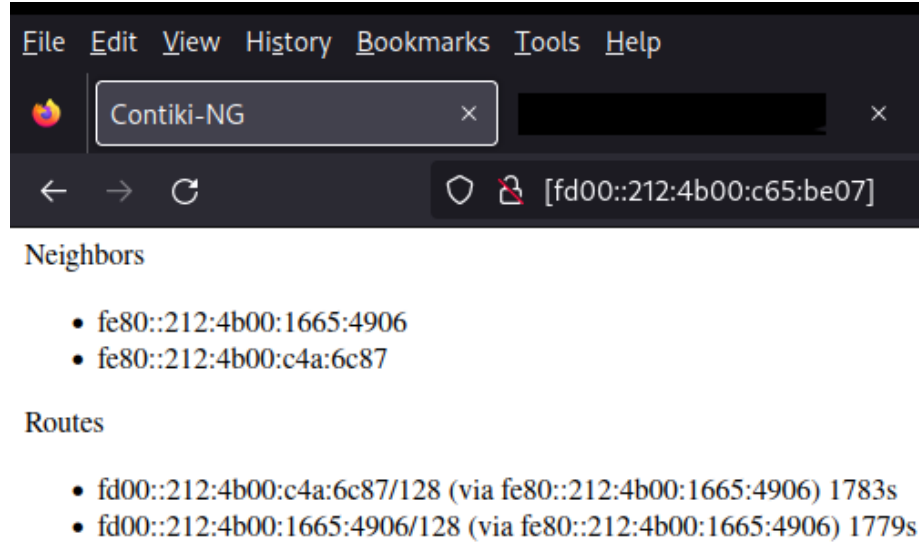


Figure 4: Enforcement of the topology.

2.3 Deploying the Attack

The attack we deploy is the *rank attack*.

In the *rank attack*, a malicious node advertises a lower rank to the target nodes, in a way to be selected as the *preferred parent* and attract all the traffic. Specifically, in this case, such a rank is lower than the ones of all involved nodes, except for the border router.

One of the ways to realize the *rank attack* consists in setting the rank of the malicious node lower than other nodes in the Wireless Sensor Network (WSN). To do so, the malicious node needs to know only the rank of the border router, to adapt it to its advantage.

To implement the attack in the *contiki-ng*, we first need to identify on the IoT protocol stack which layer needs to be modified. In our case, being the *rank attack* an attack to *RPL*, we need to target the files in the routing folder:


Listing 3: RPL Folder.

```
/WORKFOLDER/contiki-ng/os/net/routing/rpl-classic
```

More in detail, the following three files are interesting for us to deploy the attack:

1. *rpl-private.h*
2. *rpl-timers.c*
3. *rpl-dag.c*

In the previously listed files, we need to modify a few lines to deploy the *rank attack* statically. Note that we need to edit these files only for the malicious node.

 We remark that the following considerations apply to *RPL-CLASSIC*, and not on *RPL-LITE*. Therefore, we need to add in the *Makefiles* of each node the following line:
`MAKE_ROUTING=MAKE_ROUTING_RPL_CLASSIC.`

We start by editing the header file *rpl-private.h*. First, we set `#define RPL_MAX_RANKINC` to 0 on line 180. Through this setting, we prevent the rank of the malicious node from increasing based on the observed traffic.

Next, we set `#define RPL_INFINITE_RANK` to 256 on line 195. Through this setting, in case the rank is increased for any reason, it is limited to the value 256. The reason why this value is set to 256 will be clear later on, when we will observe the regular network topology. We provide a small excerpt of the edited *rpl-private.h* file:

Listing 4: *rpl-private.h*

```
#ifndef RPL_CONF_MAX_RANKINC
//#define RPL_MAX_RANKINC (7 * RPL_MIN_HOPRANKINC)
#define RPL_MAX_RANKINC 0
#else /* RPL_CONF_MAX_RANKINC */
#define RPL_MAX_RANKINC RPL_CONF_MAX_RANKINC
#endif /* RPL_CONF_MAX_RANKINC */
```

```

#define DAG_RANK(fixpt_rank, instance) \
    ((fixpt_rank) / (instance)->min_hoprankinc)

/* Rank of a virtual root node that coordinates DAG root nodes.
   */
#define BASE_RANK 0

/* Rank of a root node. */
#define ROOT_RANK(instance) (instance)->min_hoprankinc

// #define RPL_INFINITE_RANK 0xffff
#define RPL_INFINITE_RANK 256

```

Next, we need to edit the file *rpl-timers.c*. Here, we need to comment out the function *rpl_recalculate_ranks()* at the line 94. By commenting the function, we prevent the malicious node to recalculate the rank, because the rank of the malicious node should be fixed.

Last, we edit the file *rpl-dag.c*. In this file, we fix the rank of the malicious node. On lines 182 and 186, we return the static rank instead of the recalculated rank:

Listing 5: *rpl-dag.c*

```

rpl_rank_t
rpl_rank_via_parent(rpl_parent_t *p)
{
    if(p != NULL && p->dag != NULL) {
        rpl_instance_t *instance = p->dag->instance;
        if(instance != NULL && instance->of != NULL &&
            instance->of->rank_via_parent != NULL) {
            //return instance->of->rank_via_parent(p);
            return 129;
        }
    }
    //return RPL_INFINITE_RANK;
    return 129;
}

```

After editing all the files, we need to recompile the *hello-world* program of the edited *contiki-ng* folder.

To verify the correct execution of the code, it is necessary to plug in: (i) the border router; (ii) the sniffer, (iii) the *Simple Node 1*, (iv) the *Simple Node 2* and, at last, (v) the *Malicious Node*. Fig. 5 shows that the traffic originating from the node whose MAC address ends with *6c87* is routed to the border-router through the node with the MAC address ending with *c300*, which is the malicious node we deployed, instead that through the node with the MAC address ending with *4906*. Thus, the attack has been executed successfully.

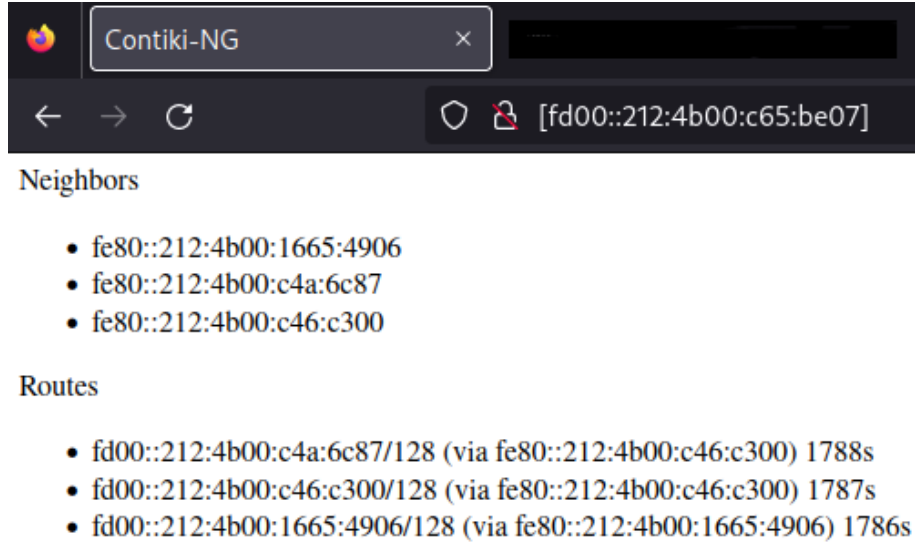


Figure 5: Malicious Node.

3 Defense Design

In this section, we design an Intrusion Detection System (IDS) as a defense mechanism against the *Rank attack* described in the previous section. We describe the corresponding hardware and software planning in Sec. 3.1, the deployment of the defense in Sec. 3.2 and, finally, Sec. 3.3 discusses the pros and cons of the proposed defense strategy.

3.1 Hardware and Software Planning

For deploying the defense against the *Rank attack*, it is necessary to have (at least) one node (in our case, *Sensortags CC2650* devices) acting as a sniffer, which should be in charge of capturing the traffic and analyzing it through *Wireshark*. Therefore, we need in total (at least) one device to realize the defense. Also, we need to save the traffic captured by *Wireshark* into a *pcap* file, and to write a script, e.g., in Python, to crawl each packet sent by each individual node to check if their rank has been recalculated.

3.2 Deploying the Defense

As outlined, we want to detect the *Rank attack* through an ad-hoc IDS.

Indeed, by analyzing the traffic exchanged between the nodes in the network, we should see that the non-malicious nodes tend to recalculate the ranks periodically while, excluding the border-router, the malicious node should have a fixed rank to be sure to carry out the attack successfully.

Thus, by analyzing the changes in the rank of the nodes in the network, we should be able to identify the malicious node and enforce the rules on the other nodes to ignore the packets originating from such a node, so as to globally repair the topology.

As shown in Sec. 2.2, we extend the code by adding the MAC address of the malicious node. The adaption is made for every node in the topology (see Fig. 1).

In Figures 6, 7, 8, 9, we show that the two legitimate nodes *Simple Node 1* (MAC address ending with 4906) and *Simple Node 2* (MAC address ending with 6c87) are recalculating their ranks periodically, while the malicious node, shown in Fig. 10, is not recalculating the rank. Thus, we can use these pieces of information to detect the anomaly and exclude the malicious node from the topology, through specific rules at the MAC layer.

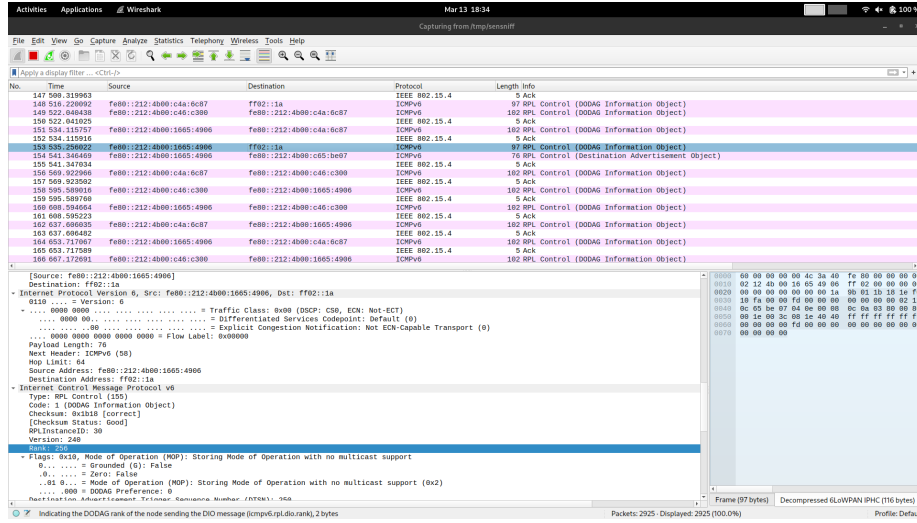


Figure 6: Rank of *Simple Node 1* (MAC address ending with 4906) at the time $t=0$.

3.3 Discussion

The proposed defense strategy works only when the rank of the malicious node is statically set, so as not to change over time. A smarter attacker might change periodically its reported rank in a narrow interval, e.g., [128 – 130], so as to avoid detection while still being able, with overwhelming probability, to carry out the attack. On the one hand, an IDS might be easily re-designed to catch such small fluctuations. On the other hand, being aware of the deployment of such IDS, the adversary might enlarge the interval of the rank to escape detection. Such a behavior would likely decrease the chances of success by the attacker, mitigating it to a limited extent.

