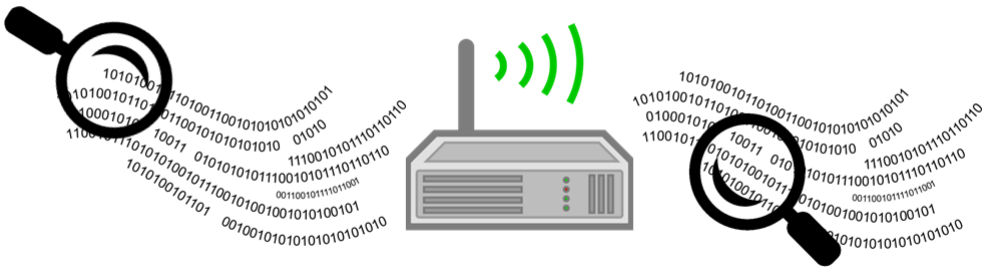
Covert Channel Defense

Cross-Router CC Manual



Oren Shvartzman

November 26, 2022

Contents

1 Introduction

1.1 Cross-Router CC - Theory of operation

The Goal of this attack, is to overcome logical network isolation (VLAN). The victim's network architecture is two computers, each connected to the same router but are in different VLANs (described in Figure 1).

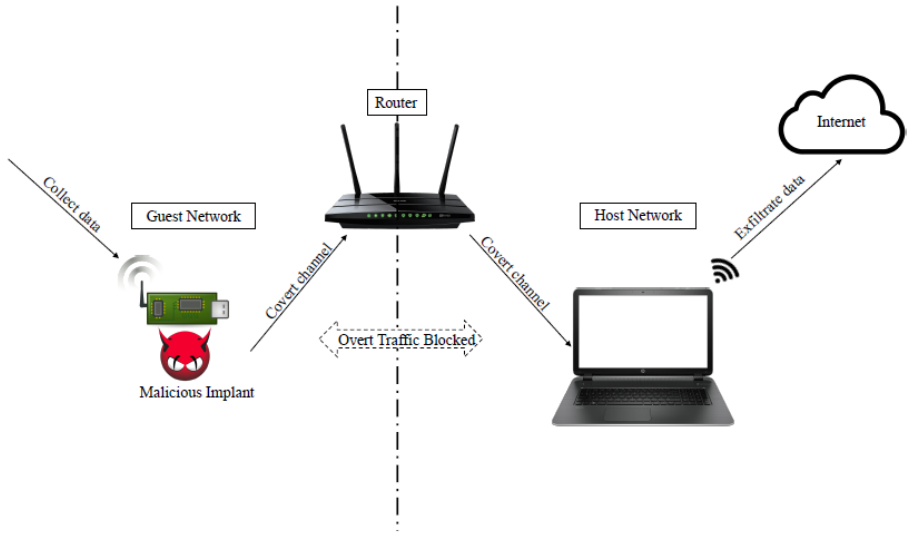


Figure 1: Cross-Router CC victim system architecture

Now, some terminology:

- The "sender" is the victim network from which the classified information is sent (using the covert channel).
- The "receiver" is the host network from which the classified information is sent to the attacker (using the internet).

The attack method is as follows. First, it is assumed that there is a malicious device implanted in the victim's network that collected classified data.

To leak the information, both networks must be synchronized in their actions. So, starting in a pre-determined time, the receiver sends large amounts of packets (could be ARP\DHCP requests or other types of packets) and measures the response time from the router. This is saved as the router's "normal" response time.

Simultaneously, the sender will do one of two actions:

- To send the bit '1', it would send large amounts of packets for T seconds.
- To send the bit '0', it wouldn't send anything for T seconds.

The receiver measures the average response time in T-seconds windows. If the average response time from the router is larger than usual - '1' was sent. otherwise, '0' was sent.

1.2 CRCC.py script

The attack is operated using the python script CRCC.py.

The goal of this program is to allow doing the attack fully. Meaning, the same script can be used for both sender and receiver sides.

This section explains in detail how the script works.

1.2.1 Script parameters

The script parameters are configured in a JSON file called: CRCC_conf.json.

The file name can be changed but must always contain the name mentioned above as a prefix (and end with .json suffix).

For example: CRCC_conf_1.json or CRCC_conf_sen1000.json will be acceptable, but 1_CRCC_conf.json or CRCC_conf.txt will not be acceptable.

The program's parameters:

- ROLE - This parameter determines the role of the RPI during the attack - sender\receiver. Can be one of two values: SEN or REC.
- TYPE - This parameter determines the packet type that will be sent. Can be one of two values: ARP or DHCP.
- RATE - This parameter determines the packet rate that will be sent. The rate will be in pps (packets per second).
- WIN_LEN - This parameter determines the length of the T-second window used by the receiver to decode the data. The time should be an integer in milliseconds.
- DATA_PATH - This parameter determines the path to the file that contains the data that should be send (the "classified" data).
- OUTPUT_PATH - This parameter determines the output file path. If the path includes a non-existant folder, it should not create a new one.

How should one configure CRCC correctly?

CRCC.py is operating well in very specific configurations.

In order to get the parameters "just right", the user must run the script with different RATE, WIN_LEN and TYPE and see if the data is transmitted correctly.

Also, the user should pay attention to the "bit_threshold" and "start_index_threshold" variables within the receiver code to decode the information correctly.

After much trial and error, the following configuration is recommneded (note that bit_threshold and start_index_threshold are found in the receiver code, but are influenced by the sender's configuration):

Receiver				
TYPE	RATE	WIN_LEN		
ARP	5	2		
Sender				
TYPE	RATE	WIN_LEN	bit_threshold	start_index_threshold
ARP	500-510	2	np.mean(median_list[-6:]) + 0.0003	2.5*normal_avg
ARP	520-560	2	np.mean(median_list[-6:]) + 0.001	2.5*normal_avg
ARP	570-600	2	np.mean(median_list[-6:]) + 0.003	2.5*normal_avg

Figure 2: REC-ARP–SEN-ARP recommneded configuration

Receiver				
TYPE	RATE	WIN_LEN		
DHCP	5	2		
Sender				
TYPE	RATE	WIN_LEN	bit_threshold	start_index_threshold
ARP	450-460	2	np.mean(median_list[-6:]) + 0.001	1.5*normal_avg
ARP	470-480	2	np.mean(median_list[-6:]) + 0.002	1.5*normal_avg
ARP	490-570	2	np.mean(median_list[-6:]) + 0.003	1.5*normal_avg
ARP	580-600	2	np.mean(median_list[-6:]) + 0.005	1.5*normal_avg

Figure 3: REC-DHCP–SEN-ARP recommneded configuration

Receiver				
TYPE	RATE	WIN_LEN		
DHCP	5	2		
Sender				
TYPE	RATE	WIN_LEN	bit_threshold	start_index_threshold
DHCP	120-140	2	np.mean(median_list[-6:]) + 0.0008	1.5*normal_avg
DHCP	150-160	2	np.mean(median_list[-6:]) + 0.001	2.5*normal_avg
DHCP	170-190	2	np.mean(median_list[-6:]) + 0.02	2.5*normal_avg
DHCP	200	2	np.mean(median_list[-6:]) + 0.04	2.5*normal_avg

Figure 4: REC-DHCP–SEN-DHCP recommneded configuration

Receiver				
TYPE	RATE	WIN_LEN		
ARP	5	2		
Sender				
TYPE	RATE	WIN_LEN	bit_threshold	start_index_threshold
DHCP	150-160	2	np.mean(median_list[-6:]) + 0.001	2*normal_avg
DHCP	170-190	2	6*normal_avg	2*normal_avg
DHCP	200	2	16*normal_avg	2*normal_avg

Figure 5: REC-ARP–SEN-DHCP recommneded configuration

The lines that are marked with light green are considered very stable configurations.

1.2.2 Basic operation

The user should prepare two configuration files for both sides of the attack: the sender and the receiver.

To start the script, the user will issue the following command on the receiver side (the computer that runs the receiver):

```
python CRCC.py [config_file_path]
```

Then, a message in the receiver terminal will be saying that the sender can be started. After this message shows, start the sender with the same command from above.

Both sides will stop by themselves.

1.2.3 Receiver start stage

In the first stage of operating the script, the receiver should be started. The receiver sends 50 packets to the router and measures the response times (the packet

type is determined by TYPE). Then, it'll calculate the average and print it to the screen.

Finally, it'll wait to the end of the current minute and print to the screen:

```
Done measuring normal responses.
```

```
Starting to record and measure.
```

```
Please start the Sender.
```

After this, the receiver starts sampling the response time (by sending packets to the router) at a constant rate (given by RATE). Also, tcpdump runs in the background and records its network traffic.

The receiver was programmed in such a way that it'll always finish after the sender, without the need of synchronization (when implmenting the receiver, the time that takes the sender to run is measured. Then, this time plus 1 minute is the time the receiver will run (hard-coded)).

1.2.4 Sender start stage

After the receiver instructs the user to start the sender, it should be started (it is done manually by the user).

As explained in the Theory of operation section, when the sender wishes to send '1', it sends loads of packets for WIN_LEN seconds. If it wishes to send '0', it does nothing.

The sender will send all of the information found in the DATA_PATH file in packets according to the CRCC packet frame (see section 1.2.5). In the current of CRCC, the payload that is sent is a constant bit series: [1, 0, 1, 1, 0, 1, 0].

1.2.5 CRCC packet frame

This is a basic packet frame designed to help the receiver decode the message correctly.

The packet frame structure is as follows:

Preamble	Payload size	Payload	Suffix
10101011	0100	1101	11111

Figure 6: CRCC packet frame

As seen in Fig. 6, the frame consists several fields:

- Preamble: a constant series ('10101011') at the start of the frame that is used to help the receiver know where the packet starts. The series '10101011' is used in the Etherenet protocol for the same purpose.
- Payload size: the size of the packet payload. Its size is always 4 bits (can be changed). The value in Fig. 6 is only an example.
- Payload: the packet's payload. This is the actual data that is leaked. The value in Fig. 6 is only an example.
- Suffix: a constant series ('11111') at the end of the frame that is used to help the receiver know where the packet ends. Also, if the suffix isn't right (not enough ones, having '11101', etc.) indicates that the receiver may divided the time windows wrong and should redo it.

The bits that are shown above are not the actual bits that are transmitted. Each bit in the packet frame is Manchester-encoded in the following manner:

- bit '1' turns to '10'
- bit '0' turns to '01'

This encoding was chosen to reduce errors during transmission.

1.2.6 Receiver decode and parse stage

After both sides are done transmitting, the receiver is left with an array of lots of response times it sampled. These response time should now be decoded to bits.

First, the receiver looks for the first abnormaly high response time (the threshold to determine this is the variable "start_index_threshold" found in the receiver code). This response indicates the start of the sender's transmittion, thus the start of the preamble. CRCC now divides time windows from there and starts decoding.

Each time window consists RATE * WIN_LEN responses. The receiver is calcualting the median of each window and compares it to the threshold determined by the "bit_threshold" variable. If it's above the threshold, it's considered a '1'. Otherwise, its a '0'.

In the current version, it extracts the payload and calculates BER against a known predetermined payload.

1.2.7 Script output

After the script is done, the output folder will contain subfolders which are named as the date and time the recording started (for example: 20.1.20_14:34:28).

Each subfolder will contain three files:

- A pcap file which is the sender or receiver's recording.
  It'll be named in the following format:
  ROLE_TYPE_RATEpps_WIN_LENms_data_filename.pcap.
- A text file that contains the BER.

The pcap files from both sides can then be merged and used for the ML detection algorithm.

2 Setup

2.1 General setup idea

The setup components are: 2 RPI, a router and a test harness computer (as shown in Figure 7):

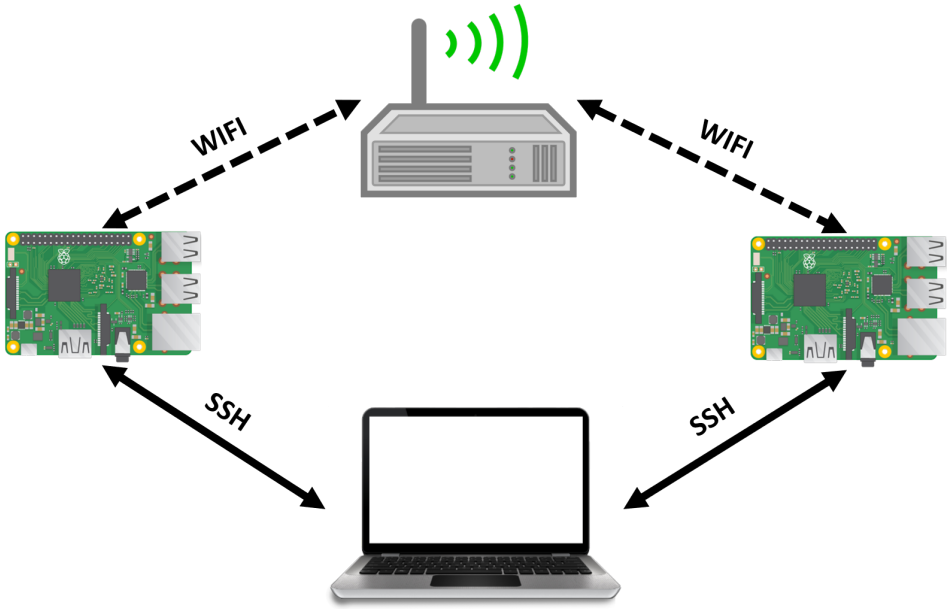


Figure 7: Cross-Router CC setup

The two RPI are connected to the router via WiFi. The RPI cannot send packets to each other, because they're connected to different WiFi LAN networks.

Both RPI are controlled remotely by the test harness computer, from which we log to the RPI using an SSH session and operate the attack script, CRCC.py.

2.2 Setup assembly

- RPI "pi-host": RPI 3 Model B, Rev 1.2. This'll run the CRCC.py script. Connected to the test harness computer via ethernet. Used as the receiver (not mandatory).
- RPI "pi-guest": RPI 3 Model B, Rev 1.2. This'll run the CRCC.py script. Connected to the test harness computer via ethernet. Used as the sender (not mandatory).

- WiFi router: D-Link-DIR-825ACG1 router. Connected to both RPI via WiFi. But, each RPI is connected to a different LAN:
  - pi-host: connected to the router's host network.
  - pi-guest: connected to the router's guest network.
- Test harness computer: the user's computer. Controls the RPI.

2.3 How to use the setup

1. open two terminals - login to each RPI
2. configure each RPI by editing CRCC_conf.json (found in `/home/pi/CRCC_src`):
   - ROLE = "REC" or "SEN" - This is the role of each RPI: sender or receiver. If one RPI is the sender, the other one must be the receiver.
   - TYPE = "ARP" or "DHCP" - This is the type of packet the RPI will send.
   - RATE = integer - This is the packet rate in pps (packet per second).
   - WIN_LEN = integer - This is the time window length in msec that the receiver uses when decoding the data.
   - DATA_PATH = `/input/data/path` - Relevant only to the sender (the receiver will ignore this). This is the path to the data it'll send (the leaked data).
   - OUTPUT_PATH = `/output/data/path` - The path to the recorded network traffic file (pcap file) will be saved.
3. After configuring both RPI, run the following command on the receiver side: `sudo CRCC.py CRCC_conf.json`
4. After it's printed on the receiver terminal you can start the sender, go to the sender side and run: `sudo CRCC.py CRCC_conf.json`
5. The sender will finish first. Go to the receiver terminal and view the BER (hope its good..).

This run will result two pcap files in the output folder (in each of the RPI). These pcap files are the network traffic recording.

Now, you should take both files and merge them using Wireshark tools:

1. Use the editcap tool with -t option to fix the time in the pcap files (the RPIs clock are not synchronized. Get the delta between them and fix one of the recordings).
2. Use the mergecap tool to merge both pcap files.

This will be the data for your ML detection algorithm.

3 Useful Commands

3.1 Useful commands

- `scp ./file.txt pi@pi-host:/home/pi` - copies file.txt from local machine to pi-host's home directory.
- `scp pi@pi-host:/home/pi/file.txt .` - copies file.txt from pi-host's home directory to local machine.
- `tcpdump -i wlan0 -s 65535 -w test.pcap` - records network traffic over wlan0 and writes to file.pcap.
- Guide for RPI to login to WiFi via command line: `https://www.raspberrypi.org/documentation/configuration/wireless/wireless-cli.md`