On Practical Aspects of Network Steganography

Jakub Oravec, Ľuboš Ovseník and Zuzana Liščinská

Department of Electronics and Multimedia Communications,
Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Němcovej 32, 040 01 Košice, Slovakia
jakub.oravec@tuke.sk, lubos.ovsenik@tuke.sk, zuzana.liscinska@tuke.sk

Abstract: This paper presents some problems with an example of a network steganography approach and proposes several simple solutions. The basic approach that uses Internet Control Message Protocol messages for injection of secret messages is demonstrated by a minimal working example. Since usage of network steganography approaches could be limited by commonly used tools such as firewalls, the paper also describes used experimental setup. Effects of presented problems and proposed solutions are analyzed by performing packet dissections in Wireshark, observation of outputs from Terminal and using measured values of Round Trip Times. It is shown that application of the proposed solutions should hide easily detectable signs about manipulation of received Internet Control Message Protocol messages, but users with some theoretical knowledge in the field should still be able to reveal usage of steganography. The paper also mentions several other ideas for further improvement of network steganography approaches in the future.

Keywords: Internet Control Message Protocol; network steganography; packet manipulation

1 Introduction

Steganography can be viewed as an art of hiding secret messages by inserting them into other communication in an unsuspicious way. This way of information hiding can have multiple forms – in the past seemingly unused wax tablets carried secret messages on the wood under the wax, later invisible inks or texts reduced in size to fit into punctuation marks were used [1]. The arrival of modern computers created many new steganographic techniques, at first mainly approaches that use raster images for covering presence of secret messages. In the late 1990s, some techniques that use features of network protocols were introduced [2, 3].

Several other network steganography techniques were designed and presented during the 2000s and early 2010s [4-10]. In general, these were rather complex tools since some of them had even more functionality than establishing and using of a covert channel. For instance, a tool called Ping Tunnel could create tunnels

between two devices by an injection into Internet Control Message Protocol (ICMP) Echo Request and Echo Reply messages [4, 11]. This feature could be used for sending secret messages in an established covert channel, but also for encapsulation of other protocol data units (PDUs), which is the point of tunneling [12]. Since usage of such complex tools required some theoretical knowledge, at that time network steganography raised interest mainly in researcher community [13-17].

Development of some Python libraries such as Scapy [18] and NetfilterQueue [19] later popularized network steganography among common computer users. With these libraries, even a small amount of code could create new network steganography techniques, which is visible in amount of various tutorials for these libraries [20, 21] or approaches that use these libraries [22-25]. Furthermore, some older network steganography techniques were reimplemented using mentioned libraries so they could be used together with newer techniques in complex systems that allow choice of steganographic approach [26].

In this paper, we would like to present a minimal working example (MWE) of a network steganography approach using both Scapy and NetfilterQueue libraries based on mentioned tutorials and proposed solutions. After analyzing some properties of the script, several issues with it will be pointed out and this paper would try to either solve them or mitigate their negative effects. Therefore, the main benefits of this paper include:

- demonstration how usage of some simple network steganography tools could be revealed,
- identification of some common problems in area of simple network steganography tools (created by merging code parts from tutorials),
- solution of some of the mentioned problems by using custom approaches or those used in more complex tools from the past.

The rest of the paper is organized as follows: Section 2 presents a brief survey of similar techniques, presented either in a form of tutorials aimed for common computer users, or as scientific papers. Section 3 describes used experimental setup, analyzed network steganography approach and the proposed MWE. Section 4 points out the problems caused by simplicity of the presented MWE, includes solutions for some of the problems and illustrates their effects on the analyzed network steganography approach. The general advice for other researchers and possible plans for future research are presented in section Conclusions and Future Plans.

2 Related Work

Some of the first network steganography approaches were proposed by Rowland in 1997 [2]. The first two methods from [2] modify either ID field or initial number in Sequence field of an Internet Protocol version 4 (IPv4) packet and both are capable of hiding one byte of secret message in each IPv4 packet. The third proposed method was not evaluated in such detail as the other two.

A paper regarding analysis of network steganography by Fisk et al. from 2002 [5] also mentions a way to use urgent pointer in Transmission Control Protocol (TCP) segments for sending as much as 2 bytes of secret message in one TCP segment. An approach that measures mean delay between IPv4 packets and can increase the delays based on secret message bits was proposed by Berk et al. in 2005 [6].

A technique called RSTEG using retransmissions of TCP segments was presented by Mazurczyk et al. in 2011 [7, 10]. RSTEG uses fact that if reception of some TCP segment is intentionally not acknowledged until timeout runs off, the segment would be sent again (it would be retransmitted). The payload of this segment could be replaced by the secret message, which provides high capacity (theoretically more than 1,400 bytes).

In 2012, Gimbi et al. proposed an approach [8] that exploits the fact that source port numbers of TCP segments could be chosen from certain interval of integers. However, the interval could be platform-specific as Windows machines use other interval as Linux machines. Therefore, the capacity of this solution could not be clearly determined.

Jankowski et al. designed a technique [9] in 2013 that uses padding present in multiple network protocols. Since network protocols require different amount of padding, the capacity of this approach depends strongly on a type of transmitted network traffic.

There are also several implementations of network steganography tools that do not use approaches presented in scientific papers. These include modification of two most significant bits in Time to Live (TTL) field of IPv4 packets [22] or offline alteration of various Internet Protocol (IP) and ICMP fields [23]. Furthermore, in 2022 Iglesias et al. proposed a complex system [26] that stops the network traffic, evaluates if it could be useful for steganography, and in that case the system injects the secret message into the traffic before it is forwarded to its destination.

A practical study on effects of network steganography tools on network traffic and its properties (e. g. introduced delays and bandwidth decrease) was presented by Hospital et al. in 2021 [24]. Certain aspects of network steganography tools, such as platform-specific design, were briefly analyzed in a paper by Bistarelli et al. from 2024 [25]. Their proposal resembles behavior of a computer running Windows operating system, and comparison of these packets with packets generated by different systems could raise suspicion about system behavior.

3 Methodology

Since evaluating practical properties of various network steganography approaches might require specific system settings (e. g. usage of certain fields in protocol headers), we decided to build an experimental setup dedicated to this task. Then, we chose a network steganography approach that would be useful for demonstrating some common problems and produced a simple script that would serve as a MWE.

3.1 Experimental Setup

The experimental setup shown in Figure 1 consists of a power supply with a remotely controlled relay (part a), a junction box with four-channel relay board (part b) that powers other components, a MikroTik routerboard (part c) that serves as a switch and four microcomputers Raspberry Pi Model 3B+ (parts d, e, f and g). The last microcomputer (part g) is not powered from the same power socket as the power supply, so it should be always turned on and ready for remote connections.

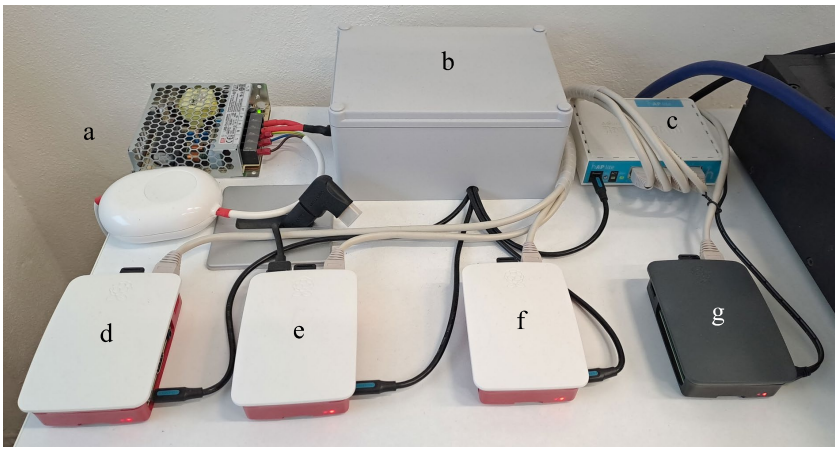

Figure 1
Used experimental setup

There are several ideas behind the used setup: it could be remotely controlled (if some of the microcomputers does not respond, it could be restarted by turning the power off and later on), one of the microcomputers is used as a server for other ones (so they run the same version of scripts) and one microcomputer is used as a 'sandbox' for testing updates before they are applied to other microcomputers.

Networks used by the experimental setup and the devices which are connected to these networks are shown in Table 1.

Table 1
Experimental networks and their devices

Network	IPv4 address range	Devices				
		.11	.12	.13	.14	other
To Internet	DHCP	no	yes	no	yes	relay server
Control (updates)	192.168.0.0/24	yes	yes	yes	no	routerboard
Experimental	192.168.1.0/24	yes	yes	yes	no	none
Internet of Things	192.168.2.0/24	no	yes	no	yes	relays
Note: words yes or no describe if device is or is not connected to certain network.						

The microcomputers (parts d to g) have last octets of their IPv4 addresses beginning with 11 and ending with 14. The first three octets in a network that connects the microcomputers to the outside world are determined by Dynamic Host Configuration Protocol (DHCP). All other networks – the Ethernet network used for controlling and updating microcomputers, experimental network for steganography (wireless, IEEE 802.11n) and Internet of Things (IoT) network (wireless, IEEE 802.11n) use statically assigned IPv4 addresses.

The experimental network filters sent and received packets by iptables firewall in order to minimize traffic flow which simplifies the analysis of captured data. Used firewall rules block mainly multicast services such as Simple Service Discovery Protocol (SSDP) or Multicast Domain Name Service (mDNS).

The microcomputers used in the experimental setup run their native 64-bit Raspberry Pi operating system based on Debian version 11 (called Bullseye), Python version 3.9.2, its libraries Scapy version 2.5.0 and NetfilterQueue version 1.1.0. Detailed packet analysis was done in Wireshark version 3.4.16.

3.2 Choice of Suitable Network Steganography Approach

Based on analysis of related work and tutorials presented on Scapy website [27], we decided to inject bytes of secret messages into a covert channel created in ICMP messages. These messages might contain a field called Data with variable size that was also used in the past for ICMP tunneling [4, 12]. Steganographic techniques that exploit this field include [9, 23, 26].

3.2.1 Internet Control Message Protocol (ICMP)

ICMP is a protocol from network layer of OSI (Open Systems Interconnection) reference model that is used for sending and receiving service messages [11]. These are used mainly for detecting connectivity between interfaces of two devices or latency measurements. ICMP has several different message types, the most common are ICMP Echo Request and ICMP Echo Reply messages. A simple example of their usage for evaluating connectivity is demonstrated in Figure 2.

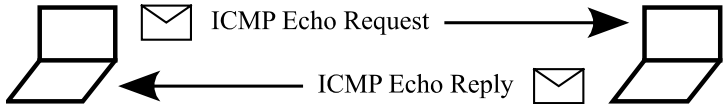

Figure 2
Usage of ICMP messages for detecting connectivity

The connectivity could be verified together with measurement of link latency by a simple utility called ping. The latency represents time necessary for delivering a pair of Echo Request and Echo Reply messages, therefore, it is called Round Trip Time (RTT). An output from Linux's Terminal that shows usage of utility ping with three pairs of messages is shown in Figure 3.

```
$ ping -c 3 192.168.1.13
PING 192.168.1.13 (192.168.1.13) 56(84) bytes of data.
64 bytes from 192.168.1.13: icmp_seq=1 ttl=64 time=3.27 ms
64 bytes from 192.168.1.13: icmp_seq=2 ttl=64 time=3.42 ms
64 bytes from 192.168.1.13: icmp_seq=3 ttl=64 time=3.56 ms

--- 192.168.1.13 ping statistics ---
3 packets transmitted, 3 received,
0% packet loss, time 2004ms
rtt min/avg/max/mdev = 3.266/3.415/3.556/0.118 ms
```

Figure 3
A Terminal output displaying Round Trip Times obtained by ping utility

ICMP messages contain several fields. It is important to mention that the presence of fields or their sizes are platform-specific since they are described in RFC 792 [11], which is only a recommendation. Fields present in ICMP Echo Request and Echo Reply messages for the experimental setup are disclosed in Figure 4.

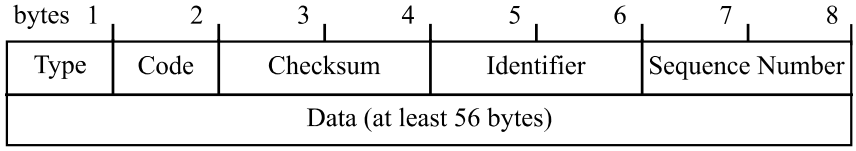

Figure 4
An example of ICMP Echo Request or Echo Reply message header, adapted from [11]

The ICMP Data field is included in Echo Request and Echo Reply messages for monitoring state of investigated connection. Echo Request messages usually use random data and Echo Reply messages should return the same data. In case when some Echo Reply message returns different data (it was either corrupted or purposely modified during transmission) or it does not arrive until timeout on sending device runs off, that connection attempt is evaluated as unsuccessful.

3.3 Minimal Working Example (MWE)

Source code of the MWE that will be used for describing potential problems with network steganography approaches is presented in Figure 5. This source code together with other ones is also available at first author's github repository [28].

```
#!/usr/bin/python

from netfilterqueue import NetfilterQueue
from scapy.all import *

def process(pkt):
  scapyPkt = IP(pkt.get_payload())

  if scapyPkt.haslayer('ICMP') and scapyPkt.haslayer('Raw'):
    icmpData = list(scapyPkt[Raw].load)
    message = sys.argv[2]

    icmpData[:len(message)] = [ord(i) for i in message]
    scapyPkt[Raw].remove_payload()
    scapyPkt[Raw].load = bytes(icmpData)
    del scapyPkt[IP].len
    del scapyPkt[ICMP].chksum

  pkt.set_payload(bytes(scapyPkt))
  pkt.accept()

os.system('sudo iptables -A OUTPUT -d ' + sys.argv[1] +
          ' -p icmp --icmp-type 8 -j NFQUEUE --queue-num 1')
print('Press Ctrl+C to end injection of secret message!')

nfqueue = NetfilterQueue()
nfqueue.bind(1, process)
try:
  nfqueue.run()
except KeyboardInterrupt:
  os.system('sudo iptables -D OUTPUT -d ' + sys.argv[1] +
            ' -p icmp --icmp-type 8 -j NFQUEUE --queue-num 1')
  nfqueue.unbind()
  sys.exit(0)
```

Figure 5
Minimal working example of analyzed network steganography approach

The MWE and all other scripts from github repository [28] need to be executed by a superuser since Scapy and iptables work directly with system kernel. The MWE uses two arguments – destination IPv4 address for stego packets with injected

secret message and character string containing the secret message. For the sake of clarity there are not any checks for these arguments in the MWE, but the scripts from [28] check for presence and format of arguments.

It is important to point out that the MWE itself does not create any cover data. Since the investigated network steganography approach uses ICMP Echo Request and Echo Reply messages as cover data, these need to be generated outside the MWE. The simplest tool for this purpose is the ping utility. After this utility creates the ICMP Echo Request messages, the MWE injects secret message into the ICMP Data field of the generated messages.

The MWE starts with a shebang and import of required libraries. Lines 6 to 20 are used for definition of processing function which begins by converting NetfilterQueue object to a Scapy packet (line 7). Then, presence of required header fields is checked (line 9), their data is converted (line 10) and secret message is injected to the converted data (lines 13 to 15). Since some of IP header and ICMP header fields are modified, it is necessary to recompute IP packet length (line 16) and ICMP checksum (line 17). The modified Scapy packet is then converted to a NetfilterQueue object (line 19) and it is forwarded (line 20).

The above mentioned function is not called at the beginning of the script's run, which starts with establishing of iptables rule for packets sent to the desired IPv4 address (line 22). These packets would be appended as objects to a NetfilterQueue queue, which is created (line 25) and the function for processing objects in the queue is specified (line 26). The queue is then started in a try block (lines 27 to 32) which stops after pressing Ctrl+C. After the queue is stopped, iptables rule is disabled (line 30), the queue is unbound (line 31) and the script could be ended (line 32).

4 Formulation of Problems and their Solutions

Experiments with the MWE shown in Figure 5 produced several problems. Corresponding versions of scripts (both problematic and those with implemented solutions) are presented at github repository [28].

4.1 Problem 1: ICMP Data Field Size

Since the recommendation for ICMP protocol – RFC 792 [11] does not clearly limit lengths of Echo Request and Echo Reply messages, they are restricted only by maximal size of frames that are used for their encapsulation. Operating system used in the experimental setup limits frame size to 1,500 bytes by default and since data fields in other headers (or a footer) usually require 42 bytes, in theory

the other 1,458 bytes are left for ICMP Data field. It might seem that all these bytes could be used for injection of the secret message.

However, ICMP Echo Request messages with this size sent to devices in the experimental setup have not produced answers in form of ICMP Echo Reply messages. This behavior is desired since it was designed as a defensive mechanism against ICMP tunneling [12]. The Echo Reply messages crucial for normal function of ICMP and displaying statistics such as RTT are sent only if Echo Request messages have so-called standard sizes, which is 98 bytes for the experimental setup.

The situation when injection of secret message into Data fields of ICMP Echo Request messages enlarges them and the receiving device does not send back ICMP Echo Reply messages is shown in Figures 6 and 7. The first mentioned figure shows a shortened packet capture from Wireshark and the second one presents a Terminal output.

```
No. Time  Src   Dst   Prot Length Info
 1  0.000 .1.11 .1.13 ICMP 1500   Echo request (no response found!)
 2  0.999 .1.11 .1.13 ICMP 102    Echo request (no response found!)
 3  2.013 .1.11 .1.13 ICMP 98     Echo request (reply in 4)
 4  2.014 .1.13 .1.11 ICMP 98     Echo reply   (request in 3)
```

Figure 6

A shortened packet capture showing ICMP Echo Reply message only after ICMP Echo Request message has certain size

```
$ ping -c 3 192.168.1.13
PING 192.168.1.13 (192.168.1.13) 56(84) bytes of data.
64 bytes from 192.168.1.13: icmp_seq=3 ttl=64 time=2.93 ms

--- 192.168.1.13 ping statistics ---
3 packets transmitted, 1 received,
66.6667% packet loss, time 2026ms
rtt min/avg/max/mdev = 2.934/2.934/2.934/0.000 ms
```

Figure 7

A Terminal output showing response only for message with certain size

Common users might not analyze network traffic via packet dissections from Wireshark, but absence of some RTT values in the Terminal output might raise some suspicion. Limiting size of ICMP Echo Request messages to the standard size of 98 bytes means that only 56 bytes per packet are available for injection of the secret message, but these ICMP Echo Request messages should be answered by ICMP Echo Reply messages. These messages would provide data for displaying RTT values in the Terminal output and since the output would have an expected form, it should not raise any suspicion about usage of steganography.

4.2 Problem 2: Dissection of ICMP Messages into Fields

Although the limitation of secret message size to 56 bytes solves the issue that was visible in previous packet capture, it still does not solve all problems. A shortened packet capture made after injecting parts of secret message with at most 56 bytes into ICMP Echo Request messages is shown in Figure 8.

```
No. Time  Src   Dst   Prot Length Info
 1  0.000 .1.11 .1.13 ICMP 98     Echo request (reply in 2)
 2  0.002 .1.13 .1.11 ICMP 98     Echo reply   (request in 1)
 3  1.000 .1.11 .1.13 ICMP 98     Echo request (reply in 4)
 4  1.001 .1.13 .1.11 ICMP 98     Echo reply   (request in 3)
 5  2.002 .1.11 .1.13 ICMP 98     Echo request (reply in 6)
 6  2.004 .1.13 .1.11 ICMP 98     Echo reply   (request in 5)
```

Figure 8
A shortened packet capture showing correct behavior of ICMP – Echo Request messages are followed by corresponding Echo Reply messages

However, another issue is visible in a Terminal output presented in Figure 9.

```
$ ping -c 3 192.168.1.13
PING 192.168.1.13 (192.168.1.13) 56(84) bytes of data.
ping: Warning: time of day goes back
(-3146830176679208565us), taking countermeasures
ping: Warning: time of day goes back
(-3146830176679202746us), taking countermeasures
64 bytes from 192.168.1.13: icmp_seq=2 ttl=64 time=0.000 ms

--- 192.168.1.13 ping statistics ---
3 packets transmitted, 1 received,
66.6667% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms
```

Figure 9
A Terminal output showing invalid timestamps in some messages

This problem is caused by a fact that RFC 792 which defines ICMP messages is only a recommendation [11]. Therefore, implementations of ICMP vary in some details and the experimental setup uses a pair of timestamps between mandatory ICMP header fields presented in Figure 4 and ICMP Data field. This is a desired behavior since it enables relatively accurate latency measurements.

For preventing any kind of suspicion and keeping the timestamps unmodified, the injection of secret message should not use 16 bytes that follow after ICMP header. This could be done quite easily, but there might be some problems with dissection of raw data into protocol fields. While Wireshark dissects the first 8 bytes of timestamps into a separate field and other 8 bytes to beginning of a field called

'Data', Scapy dissects all 16 timestamp bytes into a specific layer called 'Raw'. An example dissection of timestamp bytes in Wireshark is shown in Figure 10.

```
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x4987
    Identifier (BE): 104 (0x0068)
    Sequence Number (BE): 1 (0x0001)
    Timestamp from icmp data: Dec 28, 2024 14:35:54.000 CET
                              (0x ba fe 6f 67 00 00 00 00)
    Data (48 bytes)
0000  bf d6 05 00 00 00 00 00 10 11 12 13 14 15 16 17
0010  18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27
0020  28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37
```

Figure 10
Dissection of ICMP timestamp bytes into two different fields in Wireshark

Omitting the 16 bytes from 'Raw' layer in Scapy during injection of secret message further decreases capacity of the investigated approach to 40 bytes of secret message per one packet. Also, even when these bytes are left unmodified, there is still a minor issue regarding Terminal output. This problem is caused by packet processing in Scapy, which uses library called libpcap.

After the cover data generated by a sending device (ICMP Echo Request messages) are dissected and modified by libpcap library, a receiving device gets already modified data. It produces replies (ICMP Echo Reply messages) with modified data which are delivered to the original sending device. This means that the sending device has different values in ICMP Data fields of sent and received messages and therefore Terminal output is similar to one presented in Figure 7.

The issue with different values in Data fields of ICMP messages could be solved by a simple technique which replaces injected secret message after it was received by libpcap library on the receiving device. For purposes of this paper, we named this approach as masking technique. Its workflow is demonstrated in Figure 11.

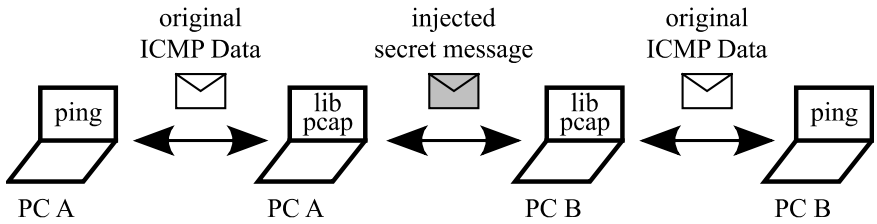

Figure 11
Masking technique employed for correcting Terminal output

Usage of the masking technique results in transmission of secret messages only between libpcap libraries of communicating devices. Since Wireshark also uses libpcap library for capturing network traffic, the injected secret messages are still visible in Wireshark packet dissections, but they are later replaced by original values from Data field. Then, the ICMP Echo Request messages are processed and answered with ICMP Echo Reply messages. The presence of injected secret message in Wireshark packet dissection is shown in Figure 12 and a corresponding Terminal output is illustrated in Figure 13. This example uses a section of 'Lorem ipsum' placeholder text [29].

```
Internet Control Message Protocol
    (some output omitted)
    Data (48 bytes)
 Data: 9c 0e 0a 00 00 00 00 00 4c 6f 72 65 6d 20 69 70
       73 75 6d 20 64 6f 6c 6f 72 20 73 69 74 20 61 6d
       65 74 2c 20 63 6f 6e 73 65 63 74 65 74 75 72 20

        .  .  .  .  .  .  .  .  L  o  r  e  m     i  p
        s  u  m     d  o  l  o  r     s  i  t     a  m
        e  t  ,     c  o  n  s  e  c  t  e  t  u  r
```

Figure 12

Presence of a secret message in a packet dissection in Wireshark

```
$ ping -c 3 192.168.1.13
PING 192.168.1.13 (192.168.1.13) 56(84) bytes of data.
64 bytes from 192.168.1.13: icmp_seq=1 ttl=64 time=116 ms
64 bytes from 192.168.1.13: icmp_seq=2 ttl=64 time=14.5 ms
64 bytes from 192.168.1.13: icmp_seq=3 ttl=64 time=13.2 ms

--- 192.168.1.13 ping statistics ---
3 packets transmitted, 3 received,
0% packet loss, time 2003ms
rtt min/avg/max/mdev = 13.241/47.826/115.700/47.996 ms
```

Figure 13

A corrected Terminal output after the masking technique was applied

4.3 Problem 3: Usage of ICMP Data Field as a Bidirectional Covert Channel

Since the transmission of secret messages requires two devices – one that injects secret messages into ICMP Echo Request messages and other device that receives these messages, extracts the secret messages and returns ICMP Echo Reply messages, it is natural to think that the created covert channel is bidirectional.

This is true in most cases, but there are also some exceptions. For instance, when PC A is connected to PC B and both PCs start injecting secret messages into ICMP messages at the same time, the resulting values in ICMP Data fields in Wireshark packet dissections of sent and received packets would be different. This is caused by a fact that both Wireshark and Scapy use libpcap and therefore Wireshark dissects ICMP Data field before the secret message is masked.

For ensuring that only one device injects secret message at certain time, a simple mechanism that checks for modifications of ICMP Data field in received messages can be employed. For purposes of this paper, we named this technique as modification check mechanism. Its workflow is illustrated in Figure 14.

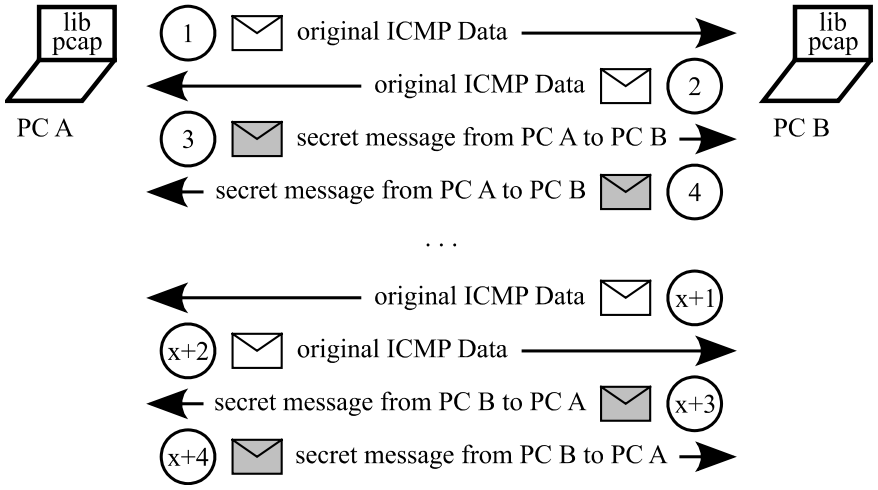

Figure 14
Modification check mechanism for detecting usage of the covert channel

The modification check mechanism starts with a step when a device that wants to inject a secret message sends an ICMP Echo Request message and then receives an ICMP Echo Reply message. If the received message carries original values of ICMP Data field, the next ICMP Echo Request message should be eligible for injection of secret message. In order to support multiple covert channels at the same time (for instance running ping commands from several devices), the received ICMP Echo Reply message and the ICMP Echo Request message which is going to be sent need to have the same values in Identifier field and subsequent values in Sequence Number field (see Figure 4 for a detail of ICMP header).

Proposed conditions rule out a situation when a device starts injecting secret message into sent ICMP Echo Reply messages. Therefore, if a device wants to inject secret messages, it has to start the communication by sending an ICMP Echo Request message.

Other drawback of the presented modification check mechanism is that it wastes one pair of ICMP messages for checking usage of the covert channel. Therefore, the capacity of investigated approach is further decreased.

4.4 Problem 4: Effects of the Proposed Solutions

While solutions proposed in previous subsections help with presented problems, they also have some negative effects on the investigated network steganography approach. First, each solution decreases capacity of the covert channel in ICMP messages, what is visible together with advantages of solutions in Table 2.

Table 2
Decreasing capacity of the covert channel after applying proposed solutions

Solution	Capacity of the covert channel per one ICMP Echo Request message	Advantage(s)
MWE	1,458 bytes	High capacity
Solution 1	56 bytes	Might be undetectable in Terminal (platform-specific)
Solution 2	40 bytes	Undetectable in Terminal
Solution 3	1 unused message and then 40 bytes	Undetectable in Terminal, supports multiple covert channels at once

Also, each added operation that processes network traffic will increase latency of the connection. The increase of latency could be measured by multiple ways, however. results of some measurements might not be useful. For instance since the injection of secret messages is done by libpcap library via Scapy, it does not make any sense to measure latency from Wireshark captures because Wireshark uses the same library and measured times would not include the processing time.

Therefore, the latency measurements could be done by getting Round Trip Times (RTTs) from Terminal output by running ping utility. RTTs represent amount of time necessary for sending an ICMP Echo Request message and receiving an ICMP Echo Reply message. Since RTTs are not shown if values in Data field of sent and received messages are different, they could not be reported for all presented solutions.

Other issue with RTTs is that the first measured time could be affected by missing destination Media Access Control (MAC) address which needs to be found by Address Resolution Protocol (ARP) messages. Hence, reported measured values do not contain first returned RTT.

As the values of RTTs could be affected by different characters in secret message, a long enough text was necessary for providing parts of used secret message. Performed measurements used Universal Declaration of Human Rights [30] which

has 10,804 bytes in total. The first 440 bytes of this document were divided into 11 non-overlapping parts with lengths of 40 bytes.

The measurements were repeated for five times to identify random delays, caused by operating system searching for updates, transmission of ARP messages, etc. Reported values include minimal RTT, average RTT calculated by (1), maximal RTT and standard deviation of the RTTs computed by (2). These values are presented in Table 3.

$$RTT_{AVG} = \frac{\sum_{i=1}^{n} RTT_i}{n} \tag{1}$$

where *i* is an index of a message in RTT measurement and *n* represents amount of messages in a measurement (all measurements used 10 messages).

$$RTT_{STDEV} = \sqrt{\frac{\sum_{i=1}^{n} (RTT_i - RTT_{AVG})^2}{n}} \tag{2}$$

Table 3
Comparison of Round Trip Times without and with modifications

Measurement	**1**	**2**	**3**	**4**	**5**
Round Trip Times for unmodified ICMP messages [ms]					
Minimal	3.43	3.42	2.27	3.44	3.41
Average	4.11	3.7	2.94	3.64	4.15
Maximal	9.47	5.3	3.66	4.36	8.39
Standard Deviation	1.79	0.54	0.55	0.29	1.51
Round Trip Times for ICMP messages modified by Solution 2 [ms]					
Minimal	13.4	7.56	13	11.6	13.2
Average	13.55	12.05	13.48	14.54	14.27
Maximal	14.5	19	14.5	24.9	18.3
Standard Deviation	0.32	3.31	0.41	3.54	1.59
Round Trip Times for ICMP messages modified by Solution 3 [ms]					
Minimal	7.86	13.5	13.6	13.5	13.5
Average	11.67	14.98	14.79	15.36	16.63
Maximal	14	20.9	19.3	20.3	30
Standard Deviation	2.69	2.53	1.83	2.2	4.84

The measured results from Table 3 show that proposed solutions increase RTTs, but the increase is reasonable. While average RTT for unmodified messages took from 2.94 ms to 4.15 ms, these values for processed messages increased to intervals from 12.05 ms to 14.54 ms (for Solution 2) or from 11.67 ms to 16.63 ms (for Solution 3). For better clarity, the increase of average RTT values after modifying ICMP messages by Solution 2 and Solution 3 is displayed also graphically in Figure 15.

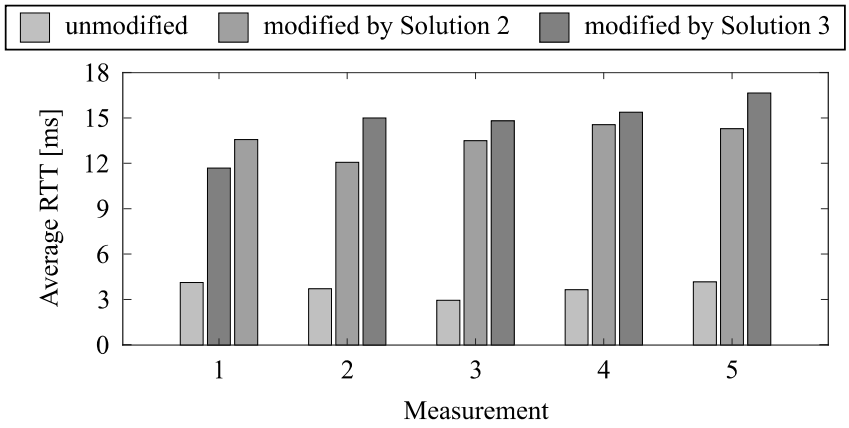

Figure 15
Increase of average RTT after performing modifications by proposed solutions

These increases of RTTs might seem quite big, but the proposed network steganography approach injects secret messages into ICMP messages used only for transmitting supplementary information during e. g. connectivity evaluation or latency measurement. Therefore, increase of RTT with size of approx. 10 ms does not cause big problems. Also, by default the ICMP Echo Request messages are generated with a frequency of one message per second, which provides a lot of time for processes such as injection or extraction of secret messages.

The measurement that scored the highest value of Standard Deviation (5th measurement for Solution 3) is further analyzed in Figure 16. It is visible that out of the 10 RTTs in the measurement, only one had a significant delay spike and the other ones were relatively close to the average value of RTT.

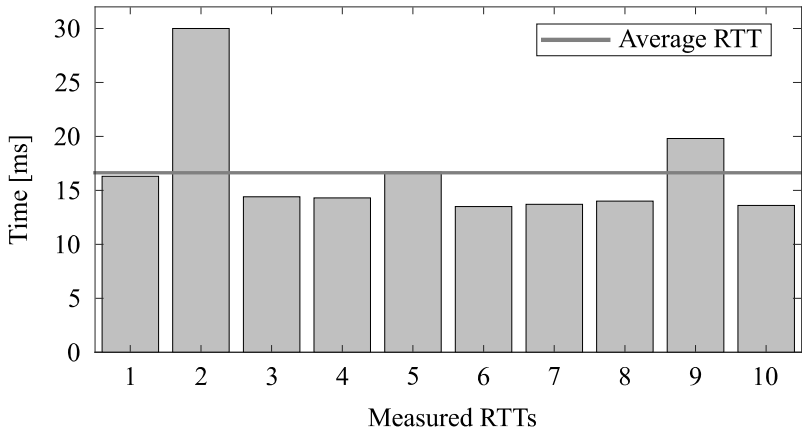

Figure 16
Variances in measured RTTs during measurement with the highest standard deviation

The mentioned delay spike can be caused by some other processes running on used device. This situation is not unusual even in dedicated experimental setups such as the one presented in section 3.1 since many operating systems are able to run multiple processes at once. Higher load on hardware or higher network utilization might increase both frequency and amplitude of these delay spikes.

4.5 Some Other Problems

There are still some well-known issues that were not solved or even analyzed in this paper. First, since both Wireshark and Scapy use the libpcap library, the secret message injected by Scapy is visible in packet dissections from Wireshark (see Figure 12). This issue could be mitigated by encryption of secret messages, however, the communicating devices would need to share encryption parameters. Since keys, nonces or other parameters have certain size, their exchange between devices might cause problems for approaches that do not create a covert channel with sufficient capacity.

Secondly, keep in mind that the analyzed network steganography approach is probably platform-specific as it was designed and tested using the experimental setup (seen in Figure 1). Different operating systems might use various implementations of ICMP protocol, which is described in RFC 792 [11]. Since it is only a recommendation and not a strict standard, ICMP messages might contain various combinations of fields. Therefore, a presence of ICMP Data field used for transmission of secret messages in this proposal might result in dropping of a packet, since the receiving device does not support presence of ICMP Data field.

Thirdly, it is quite common to block inbound ICMP Echo Request messages on firewalls since ICMP Echo Reply messages sent to various devices might reveal some information about network topology behind the firewall. In this case, the ICMP Echo Request messages could not pass through the firewall, so secret messages would not be delivered to their destination. Situations like this one limit the usability of all network steganography approaches.

Finally, some well protected networks might use an Intrusion Detection System (IDS) which is a tool that monitors inbound network traffic for anomalies and is able to create warnings. Combinations of an IDS with other tools might even block network traffic with some characteristic feature, such as suspicious values of protocol fields. In that case, the connectivity to all devices behind the IDS might be blocked which could negatively affect many more users than those which want to share secret messages by the means of network steganography.

Conclusions and Future Plans

The presented paper analyzed some practical problems regarding network steganography approaches. Currently, there is an increasing interest in this field by the community of common computer users that tend to use popular libraries such

as Scapy and NetfilterQueue. However, usage of some simple implementations might be easily detectable by noticing differences in outputs of standard tools such as Terminal. This paper proposed some solutions, but it was shown that if the analysis tool uses the same library as the processing tool (libpcap in case of Wireshark and Scapy), the analysis tool might be able to capture modified messages. Therefore, if the users have some theoretical knowledge, they could easily reveal usage of steganography.

This paper also briefly discussed possibility of encrypting secret messages prior to their injection into cover messages. This might be an interesting topic for future research since it may make evaluation of network traffic by analysis tools or an IDS more difficult. However, it places several requirements on used network steganography approach. Used approach would have to provide sufficient capacity for exchange of encryption parameters, enable their distribution in both directions, be reasonably fast and use cover messages of a common network protocol.

Also, if the used approach would not be platform-specific, it would be a big advantage over other network steganography approaches. However, a reliable declaration that the approach is not platform-specific requires either a lot of experiments on multiple operating systems or a lot of research into network protocols that have a strict structure of their fields. Both options are great choices for future work.

Acknowledgement

This work was supported by a research grant APVV-22-0400 – Extension of the autonomous applications of monitoring through multi-band UWB sensors, by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the Project No. 09I05-03-V02-00019 – Key digital technologies in wireless broadband resilient transmission networks and by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the Project No. 09I03-03-V05-00015 – Using AI in wireless broadband resilient transmission networks.

References

[1] J. Fridrich: Steganography in Digital Media: Principles, Algorithms and Applications. Cambridge University Press, Cambridge (United Kingdom), 2009, ISBN: 978-0-5211-9019-0

[2] C. H. Rowland: Covert Channels in the TCP/IP Protocol Suite. FirstMonday.org, Vol. 2, 1997, No. 5, ISSN: 1396-0466

[3] W. Mazurczyk, S. Wendzel, S. Zander et al.: Information Hiding in Communication Networks. John Wiley & Sons, Hoboken (USA), 2016, ISBN: 978-1-1188-6169-1

[4] D. Stødle: Ping Tunnel. Cited 30th December 2024, available online at: http://www.cs.uit.no/~daniels/PingTunnel/index.html

[5] G. Fisk, M. Fisk, C. Papadopoulos et al.: Eliminating Steganography in Internet Traffic with Active Wardens. Proceedings of 5th International Workshop on Information Hiding, Noordwijkerhout (Netherlands), October 2002, pp. 18-35, ISBN: 978-3-5400-0421-9

[6] V. Berk, A. Giani, G. Cybenko: Detection of Covert Channel Encoding in Network Packet Delays. Technical Report TR536, 2005, pp. 1-11

[7] W. Mazurczyk, M. Smolarczyk, K. Szczypiorski: Retransmission Steganography and Its Detection. Soft Computing, Vol. 15, 2011, No. 3, pp. 505-515, ISSN: 1432-7643

[8] J. R. F. Gimbi, D. Johnson, P. Lutz et al.: A Covert Channel over Transport Layer Source Ports. Proceedings of International Conference on Security and Management SAM '12, Las Vegas (USA), July 2012, pp. 1-5, ISBN: 978-1-6013-2230-5

[9] B. Jankowski, W. Mazurczyk, K. Szczypiorski: PadSteg: Introducing Inter-protocol Steganography. Telecommunication Systems, Vol. 52, 2013, No. 2, pp. 1101-1111, ISSN: 1018-4864

[10] W. Mazurczyk, M. Smolarczyk, K. Szczypiorski: On Information Hiding in Retransmissions. Telecommunication Systems, Vol. 52, 2013, No. 2, pp. 1113-1121, ISSN: 1018-4864

[11] J. Postel: RFC 792: Internet Control Message Protocol. Cited 30th December 2024, available online at: https://www.rfc-editor.org/rfc/rfc792

[12] A. Singh, O. Nordström, C. Lu et al.: Malicious ICMP Tunneling: Defense Against the Vulnerability. Proceedings of 8th Australasian Conference on Information Security and Privacy ACISP '03, Wollongong (Australia), July 2003, pp. 226-236, ISBN: 978-3-5404-0515-3

[13] J. Lubacz, W. Mazurczyk, K. Szczypiorski: Principles and Overview of Network Steganography. IEEE Communications Magazine, Vol. 52, 2014, No. 5, pp. 225-229, ISSN: 0163-6804

[14] S. Wendzel, W. Mazurczyk, L. Caviglione et al.: Hidden and Uncontrolled – On the Emergence of Network Steganographic Threats. Proceedings of Information Security Solutions Europe 2014, Brussels (Belgium), October 2014, pp. 123-133, ISBN: 978-3-6580-6707-6

[15] N. Singh, J. Bhardwaj, G. Raghav: Network Steganography and its Techniques: A Survey. International Journal of Computer Applications, Vol. 174, 2017, No. 2, pp. 8-14, ISSN: 0975-8887

[16] D. Galinec, W. Steingartner: Combining Cybersecurity and Cyber Defense to Achieve Cyber Resilience. Proceedings of 14th International Scientific Conference on Informatics 2017, Poprad (Slovakia), November 2017, pp. 87-93, ISBN: 978-1-5386-0888-3

[17] W. Steingartner, D. Galinec: Cyber Threats and Cyber Deception in Hybrid Warfare. Acta Polytechnica Hungarica, Vol. 18, 2021, No. 3, pp. 25-45, ISSN: 1785-8860

[18] Scapy: the Python-based Interactive Packet Manipulation Program & Library. Cited 30th December 2024, available online at: https://scapy.net/

[19] NetfilterQueue. Cited 30th December 2024, available online at: https://pypi.org/project/NetfilterQueue/

[20] crawsecurity: Packet Manipulation with Scapy. Cited 30th December 2024, available online at: https://medium.com/@crawsecurity/packet-manipulation-with-scapy-7205dec3dfdc

[21] R00tendo: Modifying Packets on the Fly (Python). Cited 30th December 2024, available online at: https://medium.com/@R00tendo/modifying-packets-on-the-fly-python-76076c5d6e1e

[22] chrispetrou: NETsteg: Network-layer Steganography. Cited 30th December 2024, available online at: https://github.com/chrispetrou/NETsteg

[23] Ocram95: pcapStego. Cited 30th December 2024, available online at: https://github.com/Ocram95/pcap_injector

[24] J. Hospital, D. Megías, W. Mazurczyk: Retransmission Steganography in Real-world Scenarios: A Practical Study. Proceedings of European Interdiscilinary Cybersecurity Conference EICC ’21, Targu Mures (Romania), November 2021, pp. 60-65, ISBN: 978-1-4503-9049-1

[25] S. Bistarelli, A. Imparato, F. Santini: A TCP-based Covert Channel with Integrity Check and Retransmission. International Journal of Information Security, Vol. 23, 2024, pp. 3481-3512, ISSN: 1615-5262

[26] F. Iglesias, F. Meghdouri, R. Annessi et al.: CCgen: Injecting Covert Channels into Network Traffic. Security and Communication Networks, Vol. 2022, 2022, No. 1, pp. 1-11, ISSN: 1939-0122

[27] Scapy Documentation: Usage – Stacking Layers. Cited 30th December 2024, available online at: https://scapy.readthedocs.io/en/latest/usage.html #stacking-layers

[28] jakuboravec71: PracNetStego. Cited 30th December 2024, available online at: https://github.com/jakuboravec71/PracNetStego

[29] Lipsum generator: Lorem ipsum. Cited 30th December 2024, available online at: https://www.lipsum.com/#Translation

[30] United Nations: Universal Declaration of Human Rights. Cited 30th December 2024, available online at: https://www.un.org/en/about-us/ universal-declaration-of-human-rights/