Covert TCP/IP network channels using Whitenoise protocol

Michal Rogala

http://www.michalrogala.com/security/whitenoise
michal.rogala@gmail.com

1. Introduction

The goal of this paper is to describe Whitenoise protocol – designed to create, manage and transfer data over covert TCP/IP channels.

Since Rowland's work [1] many publications were written about hiding data in TCP/IP stack, but each work was focused only on straightforward embedding information in protocol headers and sending it from point A to point B. Analysis of openly available publications and solutions dedicated to TCP/IP covert channels shows that all of them lack basic capabilities that allow one to use them for purposes other than proof-of-concept demonstration. Such capabilities include: error detection and correction, bidirectional communication and sessioning. So far only Nushu [2] by Joanna Rutkowska provided algorithms for safe delivery of data, unfortunately lacking all other mentioned capabilities.

The goal of creating Whitenoise protocol was to provide flexible solution for creating fully reliable, bidirectional covert network channels. Whitenoise itself is not dependent on any method of hiding data in TCP/IP, current implementation can hide data in IP ID field and TCP ISN numbers (as described later) but can easily be extended to support any new technique.

At current state Whitenoise and its implementation (as a Linux kernel module) supports:

- Using multiple data hiding techniques at one time (i.e. data can be transferred using IP ID and TCP SEQ/ISN fields simultaneously).
- Establishing and closing connection - with automatic transmission parameters negotiation.
- Integrity checks, allowing distinguishing legitimate Whitenoise packets from random network data.
- Fully bidirectional communication.
- Parallel communication with many hosts.
- Error detection and correction by data acknowledging and retransmission.
- Fully functional API for user-space applications to manage connections and transfer data over covert channels.

2. Overview of Whitenoise protocol

2.1 Protocol packets

Like any other network protocol, Whitenoise exchanges data using packets. Packets are fixed-size of 16 or 32 bits, which are default data sizes that can be safely embedded in TCP/IP packets using widely known methods of hiding data in network traffic.
Every schema presented here is in Little-Endian format. Also bits are pictured from left (lowest) to right (highest).

Whitenoise packets schemas are presented on Figures 1, 2, 3, 4 and 5:

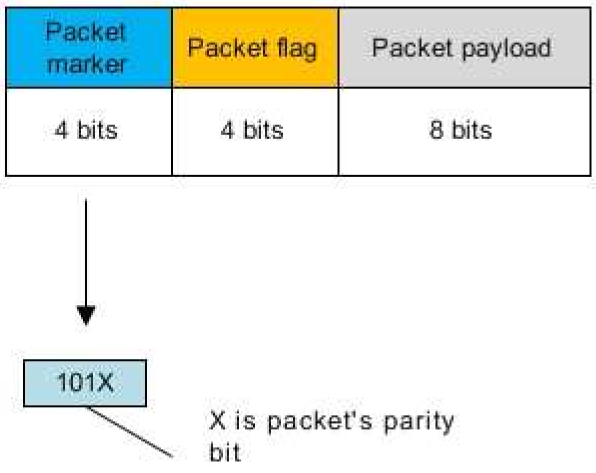

Fig. 1 – Whitenoise 16 bit control packet

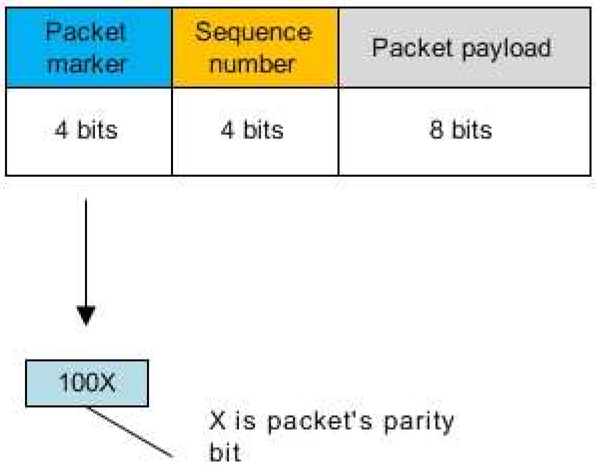

Fig. 2 - Whitenoise 16 bit data packet

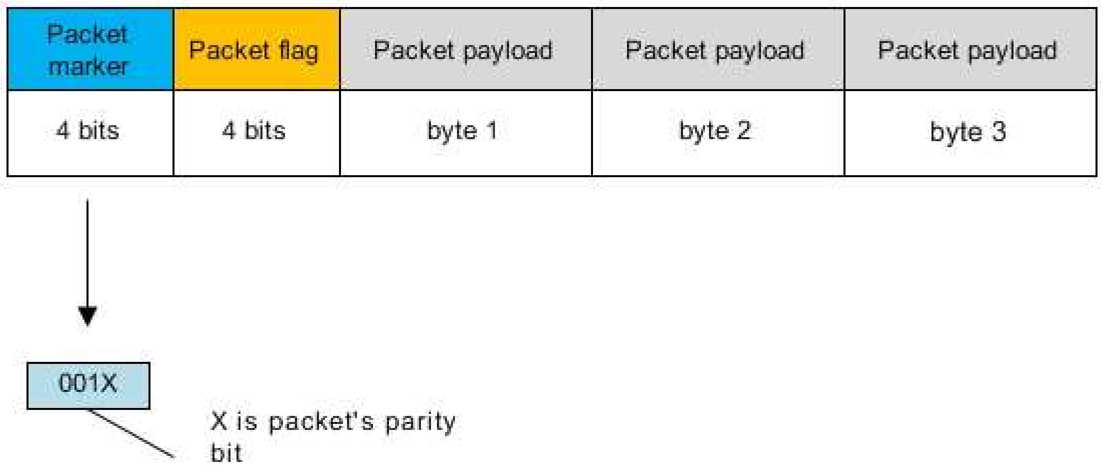

Fig. 3 - Whitenoise 32 bit control packet

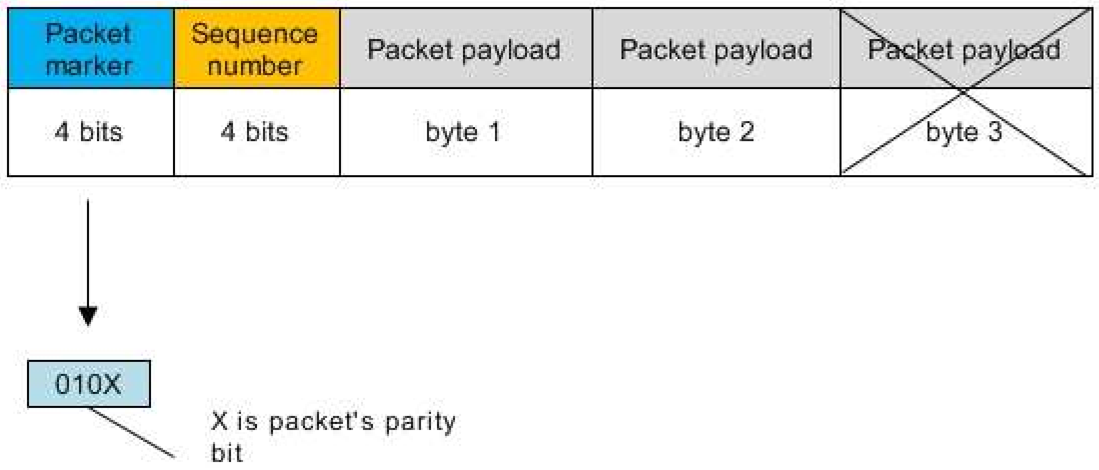

Fig. 4 - Whitenoise 32 bit data packet

Each packet starts with **packet marker** field – used both to distinguish Whitenoise packets from random network data and to check it's size (16 bit packets can be embedded in 32 bit values) and type. **Parity bit** was introduced for better discrimination of data that look like Whitenoise packet but is not.

Control packets are used to manage covert channel, while **data packets** transmit data exchanged between users. Sequence numbers in data packets are used by error detection and correction algorithms.

As 32 bit data packets can transmit either 1, 2 or 3 bytes of data, there is special need for indication how many are actually being transmitted. This problem was solved by introducing 2 types of 32 bit data packets. First, shown on Fig. 4, transmits only 2 bytes of data (3rd one is supposed to be random) while packet shown on Fig. 5 (differs by packet marker) transmits 3 bytes of data. If there is need to transmit only one byte of data at once, 16 bit packets are used.

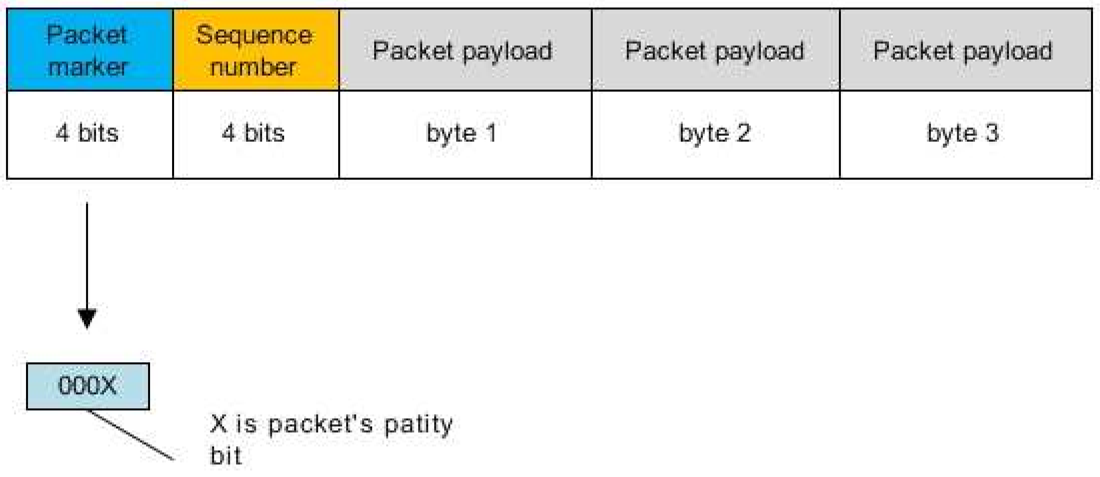

Fig. 4 – Whitenoise 32 bit data packet

2.2 Packet flags

In control packet, flags are used to determine its function. Possible packet flag values with their brief descriptions are shown in Table 1. Detailed use of each of those flags is described later in this paper.

Table 1 - Whitenoise control packet flags

Flag value (hex)	***Flag ID***	***Comment***
0x01	**INIT**	Connection initiation
0x02	**INIT_ACK**	Acknowledge of connection initiation
0x03	**INIT_ACK2**	Second acknowledge of connection initiation
0x04	**NEW_BLOCK**	Opening new data block
0x05	**NEW_BLOCK_ACK**	ACK of opening new data block
0x06	**DATA_ACK**	ACK of received data
0x07	**CLOSE**	Connection close
0x08	**CLOSE_ACK**	ACK of connection close

2.3 Mechanisms of Whitenoise protocol

2.3.1 Packet validation – parity bit

To allow better distinguishing of Whitenoise packets from pseudorandom values, parity bit was placed into packet marker field. The value of this bit is 1 if number of "ones" in whole packet is odd and is 0 if this number is even. In case of packets smaller than transport size (ie. 16 bit packets embedded in 32 bit values – when last 16 bits are random) parity bit is computed from whole transport data. When counting bits in packet, we assume that value of its parity bit is 0.

This protection is not sufficient to prevent all random data to be interpreted as Whitenoise packets, but in such small packet sizes probably there is nothing more that can be implemented (MD5 could do ;). Nevertheless – due to use of some techniques (described later) in protocol implementation - bad interpretation of received data very rarely mean that transmission will be damaged.

2.3.2 Establishing connection

To establish a connection between two hosts, 3 packets need to be exchanged between them. It's quite similar to the TCP handshake. First packet is sent by station performing active initiation (client) – its control packet with INIT flag. In response, station performing passive initiation (server) sends INIT_ACK packet. Then, client station responses to server with INIT_ACK2 packet and connection is established.

During connection handshake, the most important issue is to exchange information about communication channels supported by both sides. The basic principle of Whitenoise protocol is that it should be independent from particular methods of embedding hidden data in TCP/IP headers. Since each side of the transmission can support (or want to use) different sets of those methods, there is need to negotiate them. This is done by using payload of INIT and INIT_ACK packets where each side puts information about it's capabilities. This data is written on 8 bits – each bit of payload points to different data embedding method. Enabled (1) bit indicates that particular method is supported, disabled means that it's not. Not all bits in payload are used and those should have random values. Figure 5 presents structure of INIT and INIT_ACK packets payload.

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
IP ID	TCP ISN	reserved	reserved	reserved	reserved	reserved	reserved

Fig. 5 Structure of INIT/INIT_ACK packets payload

As you can see, there is plenty of free space in the payload – up to 8 data hiding methods can be supported in Whitenoise or in the future some bits can be used to negotiate different connection parameters.

When establishing connection, each part gets list of mechanisms supported by the other side – this list is compared with its internal list – methods present in both list are used to carry out communication. At the point of sending INIT packet, client knows nothing about mechanisms supported by the server – and therefore is supposed do sent this packet using each known method, until receiving response.

Diagram showing connection handshake and negotiation of parameters is shown on Fig. 6.

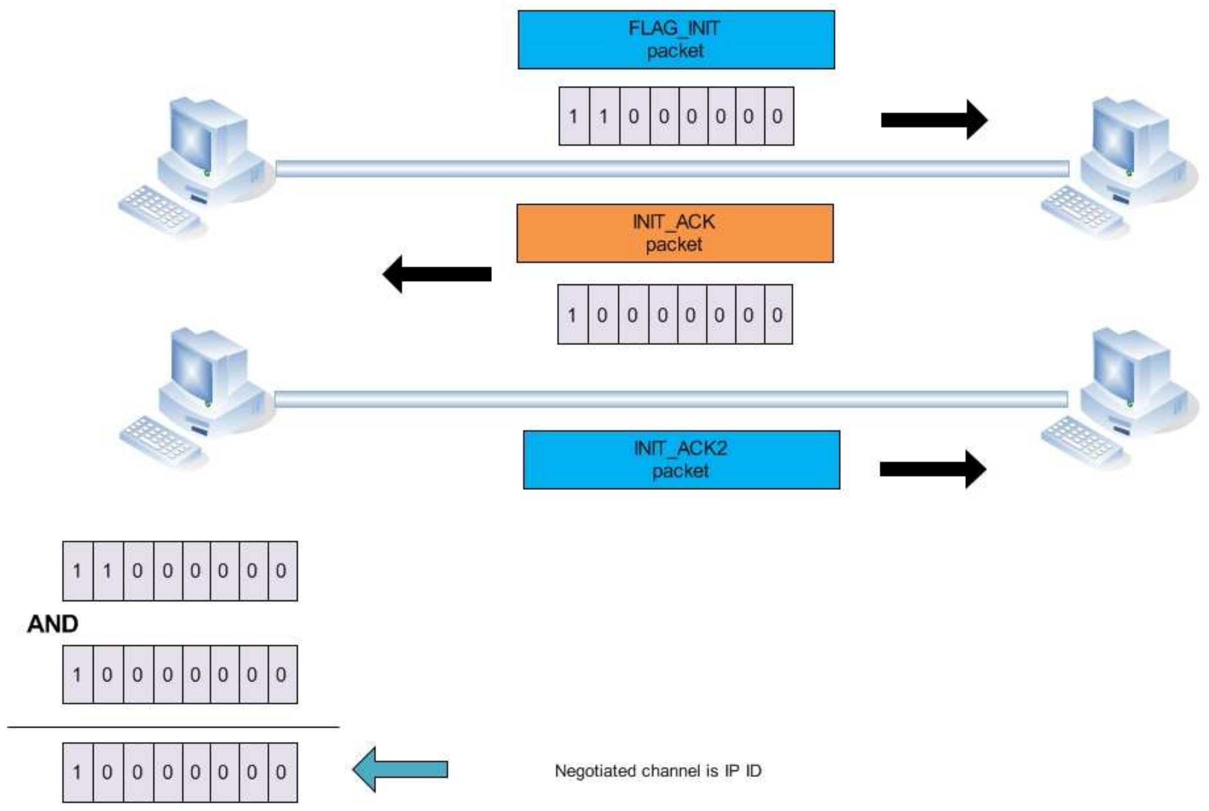

Fig. 6 Connection handshake

2.3.3 Data transmission

Having in mind, that we can store sequence number only on 4 bits, we have to group data into 16 bytes blocks. Sequencing each byte of transmitted data gives possibility to detect lost packets and retransmit them again. Each side of communication maintains two independent streams – one for sending data, second for receiving – providing bidirectional transmission.

Transmission begins when either part has data to send and indicates it by sending NEW_BLOCK packet. This opens new block of data and resets the sequence counter. It must be acknowledged by NEW_BLOCK_ACK packet. After having confirmation of successfully opened block, particular side sends data, assigning each byte a sequence number.

First 4 bits of NEW_BLOCK packet contain last number of byte to be sent in actually opened block (starting from 0), so for example setting this value for 15 mean that 16 bytes of data will be sent.

Each byte of data must be acknowledged by the receiving part. This is done using DATA_ACK packet. Payload of this packet contains sequence number of actually acknowledged byte. As sequence number is stored on 4 bits and packet payload is 8, we have 4 spare bits to use. This can be used to acknowledge many bytes of data at one time - if second value is different than 0, used with first one they indicate range of values. For example if we want to acknowledge bytes 3,4,5,6,7,8 we place numbers 3 and 8 into the DATA_ACK packet payload. Fig. 7 shows structure of this payload.

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
first sequence number				second sequence number			

Fig. 7 DATA_ACK packet payload

During transmission, data packets are sent as fast as possible – without waiting for ACK each time. When all bytes in currently opened block are sent, sending part waits for given period of time for acknowledgment of data (of course ACK packets can arrive also during transmission). When timeout occurs (more about timeouts is written later) and not all bytes are acknowledged, those bytes are sent again and again until they're ACK'ed. New data block can be opened only if all bytes from former one were successfully transmitted. Full block transmission procedure is shown on Fig. 8.

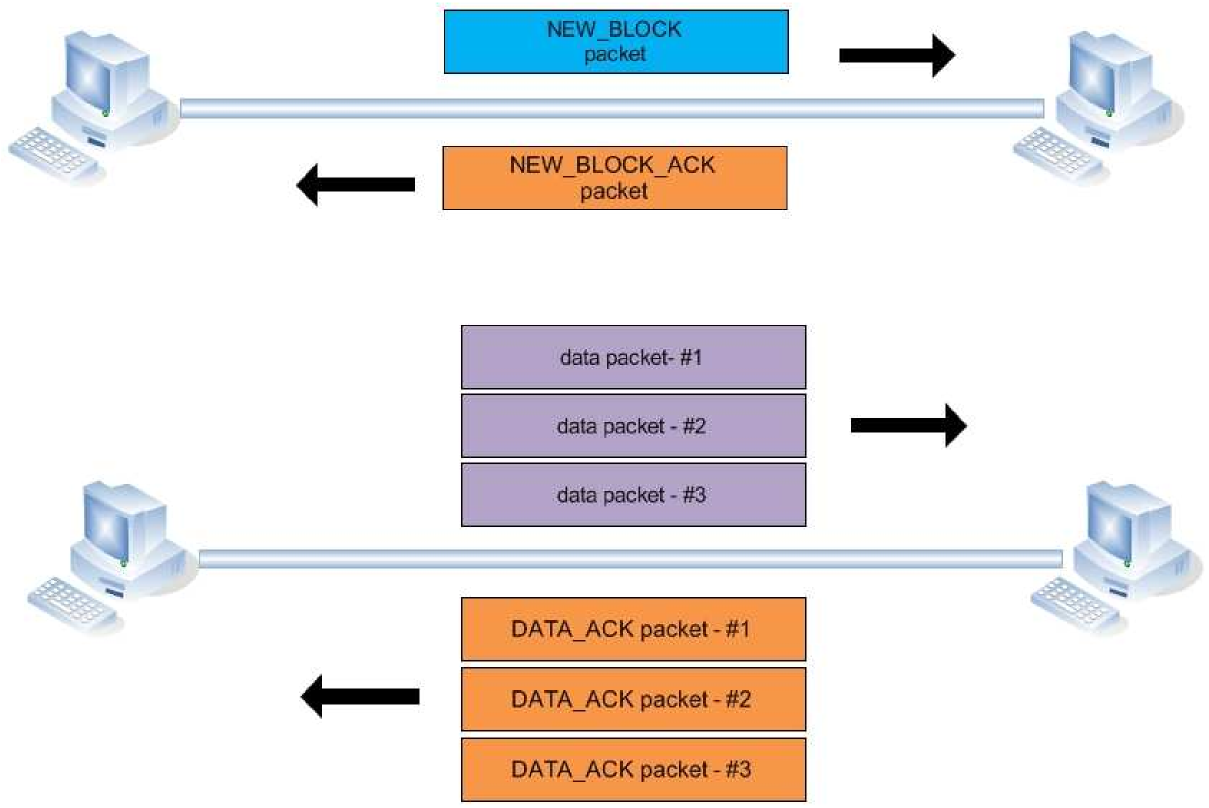

Fig. 8 Transmission of data in Whitenoise

2.3.4 Closing connection

Connection is explicitly closed by both sides of communication when neither of them has data to send. Similar as in the TCP protocol, connection can be closed only by one side of transmission – communicating that it has nothing to send but can receive data.

Procedure of closing connection is the same for each side and requires exchange of 2 packets – CLOSE and CLOSE_ACK. Transmission side which wants to close its part of connection sends CLOSE packet with 8 bit random number in payload. Receiver of this packet has to acknowledge it by sending CLOSE_ACK with the same number in its payload. This is shown on Fig. 9.

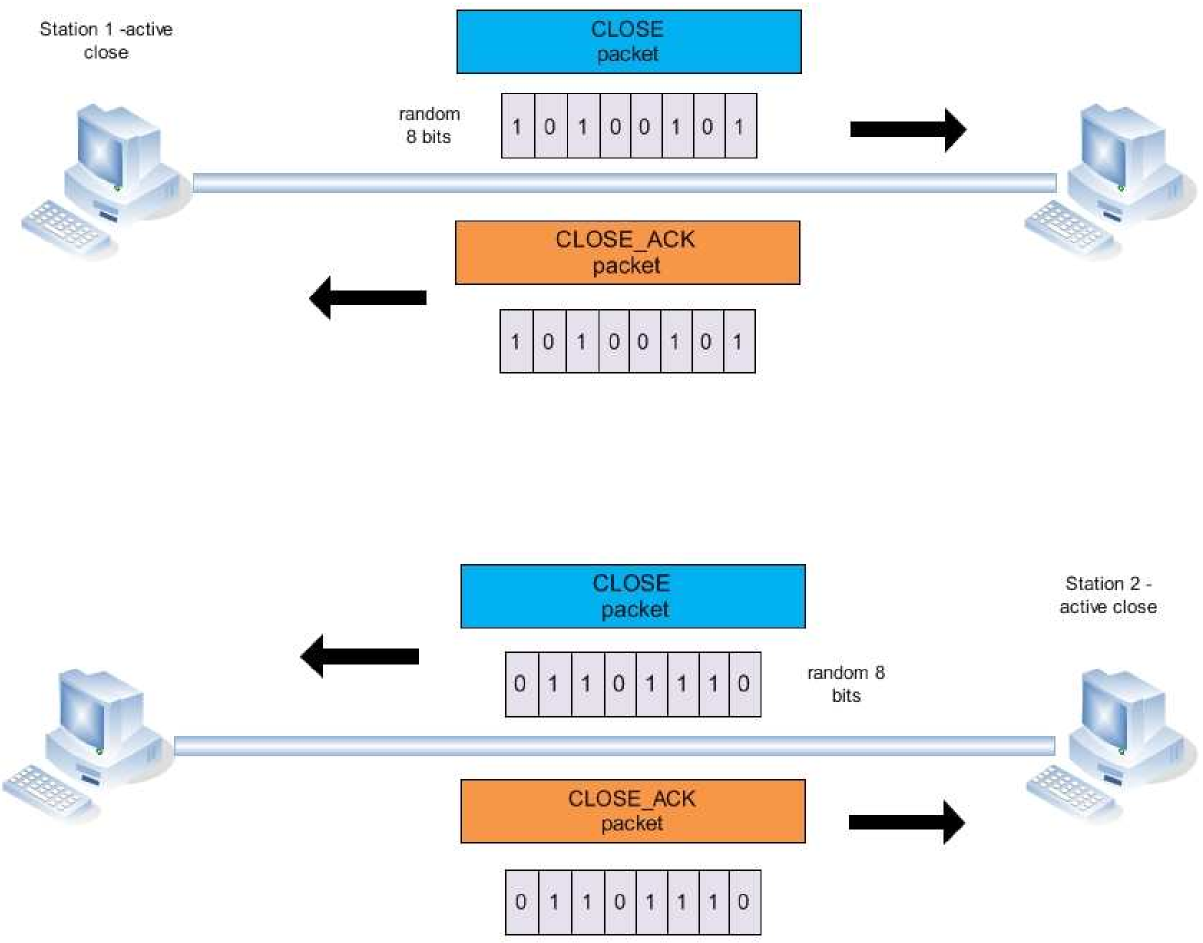

Fig. 9 – Closing Whitenoise connection

2.4 Error detection and correction mechanisms

During any operation of the Whitenoise protocol there can be situation where particular packet is lost during transmission. To prevent any damage and unspecified behavior of the protocol, error detection and correction mechanisms were implemented. Such mechanisms are very similar to those used in TCP/IP protocol and utilize state machine to check actual communication stage and possible reactions.

At the beginning, protocol is in the IDLE state. From this point user can change it either to LISTEN (when station should act as server and wait for connections) or INIT_SENT when connection should originate from the station (since setting this state Whitenoise tries to initiate connection by sending INIT packets).

After setting initial state, any sent or received Whitenoise packet changes internal protocol state as shown on Fig. 10. After each packet is sent, timeout count is set. As time measuring is pointless (TCP/IP stack can exchange packets at any rate – even few per hour, and Whitenoise can't do much about it) timeout occurs when desired response is not present in particular amount of received TCP/IP packets. After timeout – current protocol state allows to check which packet was lost and it's being sent again. Neither of the communication parts is able to check whether sent packet was lost or response to it, so part which didn't get response replays the request and the other part must response – even if this response had been already sent before. Every packet which doesn't match particular protocol state must be discarded.

Described mechanisms apply to packets which manage covert channels. After connection is established and protocol is in state ESTABLISHED_ACTIVE or ESTABLISHED_PASSIVE, exchange of user data takes place. Each independent data receiving/sending mechanism has its internal state register (which changes the way shown on Fig. 10.) and timeout counters. Detection and correction of lost packets occurs after all packets in current transmission block have been sent. After sending last packet, timeout counter is set – if all data packets are acknowledged before timeout, transmission block is considered to have been sent correctly. Otherwise – lost data packets are transmitted again until success.

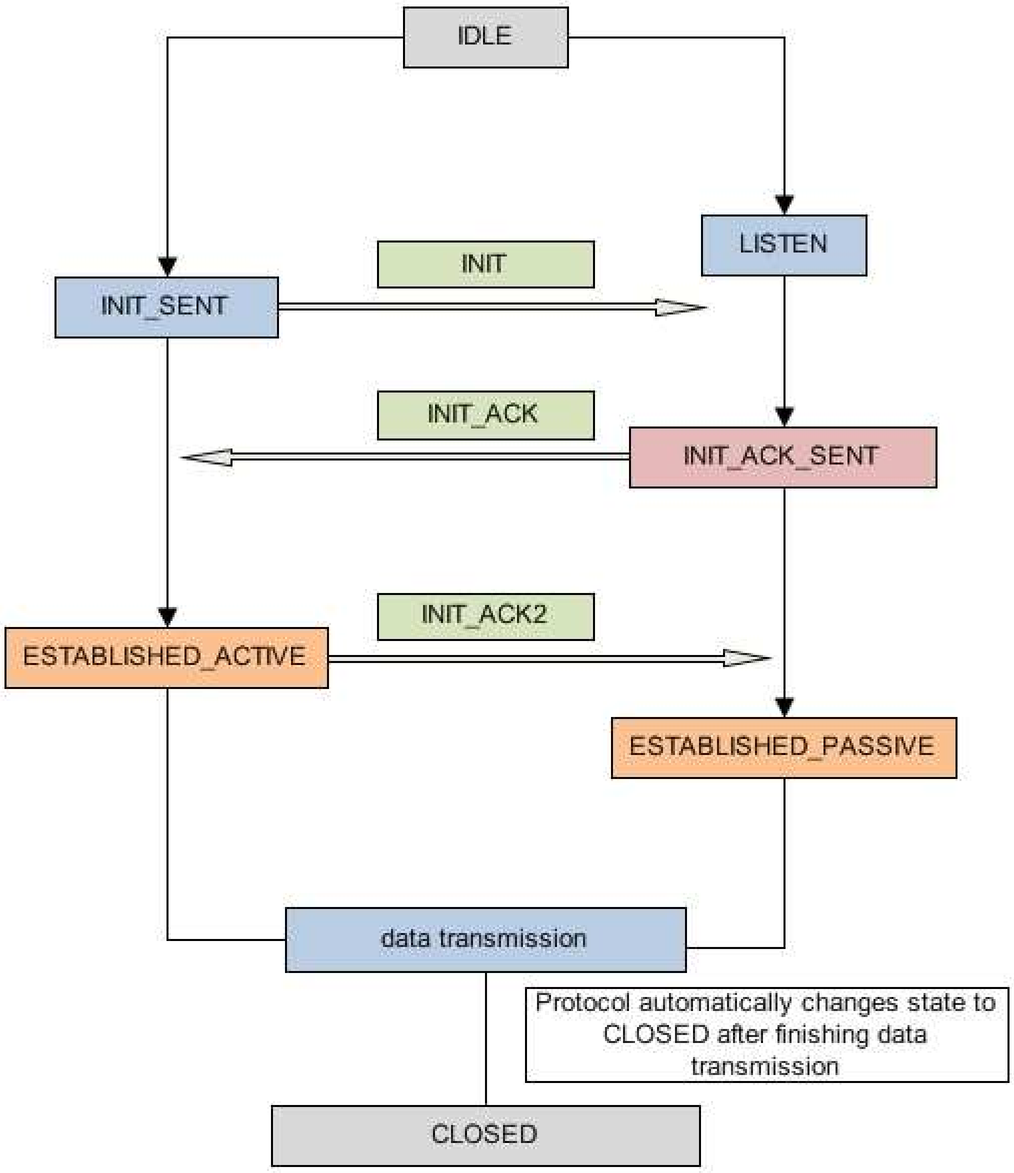

Fig 10. Protocol states and transitions between them

2.5 Protocol security

Current implementation of Whitenoise can embed its packets into either IP ID 16 bit fields or 32 bit TCP ISN fields (or use both fields simultaneously). While embedding packets into IP ID can be easily detected (value of this field usually is not random), it can be really hard to spot Whitenoise in TCP ISN fields. Usually, detecting covert channels in TCP involves checking randomness of ISN numbers – repeated or close to each other values may indicate that someone tampered the protocol (see http://lcamtuf.coredump.cx/newtcp/ for more info about evaluating randomness of TCP ISN numbers).

As Whitenoise packets are very repetitive data structures, protocol's implementation uses some techniques to make covert communication harder to spot. One of the most important features is the ability to place original ISN numbers when there is no Whitenoise packet to send (instead of some kind of predefined NULL packet). As Whitenoise packets use parity bit for simple integrity checking, then such random value won't be interpreted as protocol's packet and cause some unpredictable change to the connection. If original ISN can be interpreted as Whitenoise packet, it's lowest bit is simply changed – it's enough as countermeasure.

Another feature involves situation when some fields in Whitenoise packets are not used in particular case - then those values are set to random value, adding some additional small entropy. Below is list of fields that can be set to random values:

- Reserved bits of control packets with FLAG_INIT or FLAG_INIT_ACK flag
- Payload (8 bits) of FLAG_INIT_ACK2 control packet
- Payload (8 bits) of CLOSE control packet
- Last byte of 32 bit data packet with 010 marker (carrying only 2 bytes of transmission)
- Last 2 bytes of 32 bit value carrying 16 bit packet.

Last security feature that can be used is cryptography. At the present Whitenoise implementation uses dumb XOR mode with shared key – this is mainly to prevent detection of packets by marker, flag and parity bit. For packets to appear as true random ISN values the encryption key must be different for each one. This can be accomplished by using key initialization vector based on TCP/IP constant packet values as for example source/destination ports.

3. Implementation

The implementation code (for Linux 2.6.x) can be found at
http://www.michalrogala.com/security/whitenoise/whitenoise-release.zip

After successful loading of the kernel module, */proc/whitenoise* file should be visible. User-space applications use ioctl calls on this file to establish connection and read/write functions to transfer data through covert channel. Up to 16 simultaneous connections with different hosts can be established (this can be changed in whitenoise.h file).You can use two applications: apps/white_client.c and apps/white_server.c to transfer files between hosts – by default IP ID method is used – this can be changed in the source code of both files. More information about implementation and full Whitenoise API can be found in my thesis: http://www.michalrogala.com/security/whitenoise/ukryte_kanaly_tcp_ip.pdf

(it's in Polish – source code speaks for itself, in any other case – use google translate ;)

ATTENTION!!!!! When replacing TCP ISN numbers, internal kernel structures are modified so TCP/IP driver assumes that these are its valid numbers. So there is no need to maintain additional "proxy" kernel service that translates "fake" ISN into valid ISN and all established TCP connections work correctly after unloading the Whitenoise kernel module.

Bibliography:

[1] Rowland C.H, Covert channels in the TCP/IP protocol suite, FirstMonday - Peer Reviewed Journal on the Internet, http://www.firstmonday.org/issues/issue2_5/rowland/

[2] Rutkowska J., Passive Covert Channels Implementation in Linux Kernel, 2004, http://invisiblethings.org/papers/passive-covert-channels-linux.pdf