The Implementation of Passive Covert Channels in the Linux Kernel

Joanna Rutkowska
joanna at invisiblethings.org

Chaos Communication Congress
December 2004

Introduction

The goal of this paper is to describe the idea of so called passive covert channels (PCC), which might be used by malware to leak information from the compromised hosts. This idea has been implemented in a proof-of-concept tool, called NUSHU. The primary goal of the PCC is to be as stealth as possible by not generating its own traffic at all. To be actually useful PCC should be combined with some kind of password sniffer or other information gathering software running on the compromised host.

Idea of Passive Covert Channels (PCC)

The idea is pretty simple – we do not generate our own traffic (i.e. packets) but only change some fields in the packets which are normally generated by the compromised computer. Of course, that requires that the attacker control one of the computer which receives at least most of the traffic from the compromise host, like enterprise gateway, router, etc... For example, such passive covert channels can be used by malicious ISP employees to spy on ISP's customers.

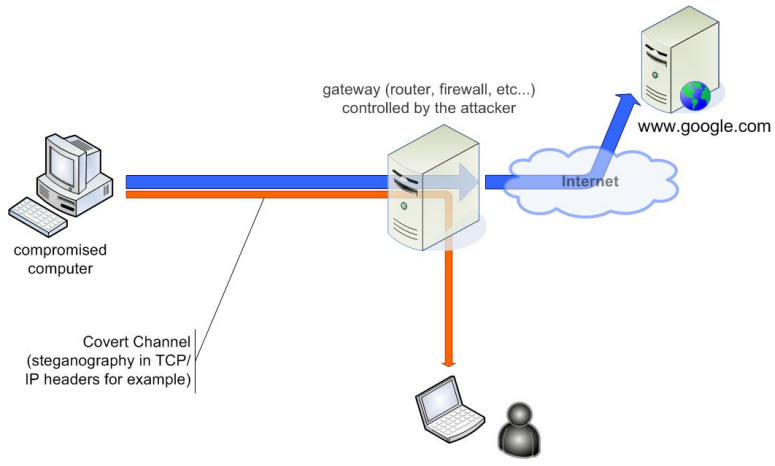

Figure 1. The idea of passive covert channel.

Although the Figure 1 shows the case of using PCC against workstation computers (which seem to make lots of requests to the internet), it should be noted, that it is also possible to use it against compromised servers (which tends to accepts requests rather then making them), see Figure 8.

How to implement

We will focus on TCP ISN based passive covert channel. There should not be much differences if we consider using a different carrier, like HTTP Cookie for example though.

There are many possible ways of implementing PCC in the Linux kernel. Two most obvious are probably the ones which exploits Netfilter hooks [4] and *ptype* handlers [3]. The exemplary implementation, described below, uses a *ptype* handler, because the author believes that this way is more difficult to detect then using an extra Netfilter hook. The PCC handler is placed on the `ptype_all`, the same list which is used by PF_PACKET sockets to place their handler, that is `packet_rcv()`.

The main task of the PCC module is to constantly change SEQ and ACK numbers between the original numbers generated by the OS kernel and the numbers which are actually transmitted overt the wire, containing the secret message (see Figure 2). If PCC changed only the first SEQ filed of the first SYN packet, then kernel would not understand the ACK number in the corresponding SYN|ACK packet which would result in breaking the connection. Also, if PCC didn't change the SEQ fields in the consecutive packets, the OS on the second end will drop that packets, because of the mismatch with the initial sequence number sent in the first packet.

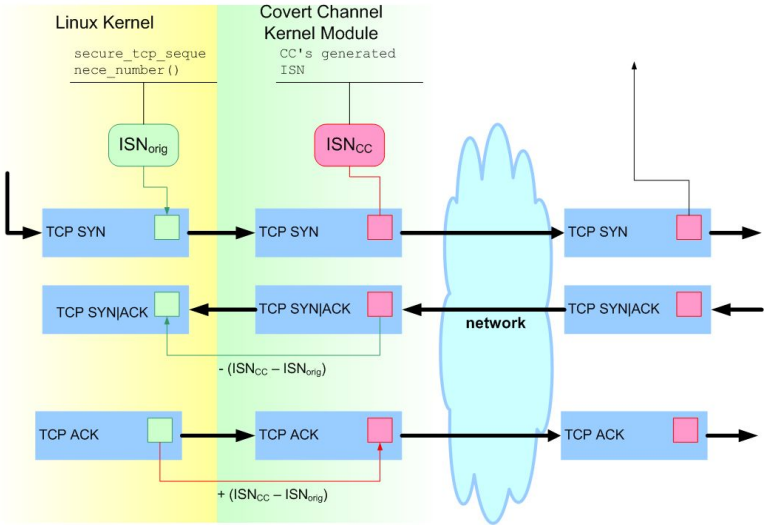

Figure 2. TCP ISN based PCC implementation overview.

To make it possible for the PCC module to change the SEQ/ACK fields as is was described above, a special structure is allocated for each new TCP connection originating from the compromised host, which holds the information about the difference between the original ISN and the one inserted by PCC (`offset` field):

```
struct conn_info {
  __u32 laddr, faddr;
  __u16 lport, fport;
  __u32 offset;   // new_isn - orig_isn
  struct list_head list;
};
```

For each packet, which is processed by PCC *ptype* handler, a check is done if it matches one of the connections stored in the `conn_info` structure. If it does its SEQ or ACK field (depending whether it is outgoing or incoming packet) is modified by the value stored in the `offset` field.

It should be obvious that PCC should also be able to detect the end of TCP connection, so it can free the unused `conn_info` structures. PCC could implement a TCP state machine to detect when the TCP connection was closed. However it seems to be much easier to periodically check the `tcphash_info` global kernel structure, which contains information about all connected sockets, and remove all `conn_info` structures which refer to non-existing connections.

Reliability layer

Any communication channel which is supposed to be used not only in the lab, but also in the wild internet, should provide a reliability mechanisms. *NUSHU*[1], the sample implementation, presented below, provides a simple protocol, which ensures the integrity of the transmitted messages as well as forcing retransmissions in the case of lost packets.

The interesting thing about this protocol is that it works with a passive receiver (i.e. the receiver does not need to change anything in the returning traffic). This is possible because the underlying protocol, i.e. TCP, already provides acknowledgment mechanism and the only thing the sender module needs to do is to recognize which TCP ACK packets match (acknowledge) the data sent over covert channel. All protocol information are sent in the SEQ/ACK fields of course, as it is depicted on Figure 3.

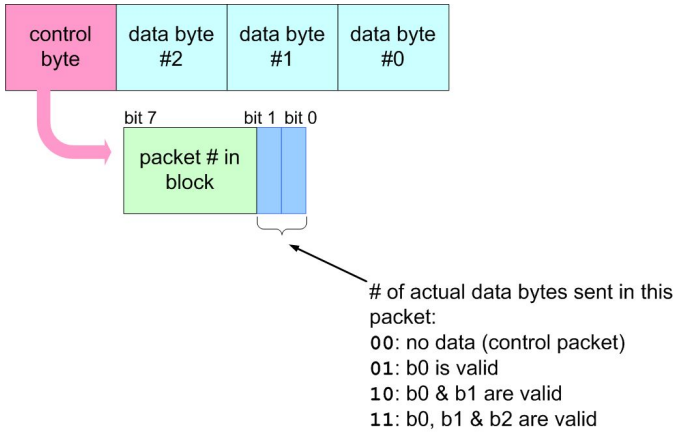

Figure 3. SEQ/ACK field layout as used by the NUSHU protocol.

Figure 4. The two special packets (ISN_EMPTY and ISN_NEW_BLOCK).

As we can see, in every TCP SYN packet, up to three data bytes can be sent. The fourth byte is used as a control byte. It carries two important information:

[1] NUSHU was a secret language developed by Chinese women hundreds years ago. Its characters were often disguised as decorative marks or as part of artwork.

- how many data bytes are actually sent in the packet (it could happen that the sender will have less then the 3 bytes waiting to be sent at the moment of TCP SYN packet is generated by the compromised host kernel)
- internal sequence number, which takes care about proper ordering of received data as well as is used for tracing which secret data TCP ACK packets actually acknowledges. NOTE: the receiving end (i.e. the OS which generates the TCP ACK packets) is usually *not* the NUSHU receiver, since NUSHU receiver is typically located somewhere in the middle (like on the corporate gate) – see Figure 1.

Because the sequence number space in this protocol is just few bits, the protocol groups data in *blocks*, takes care that all data sent within one block are acknowledged, retransmits data which were not acknowledged and then sends a special packet (ISN_NEW_BLOCK), which resets the sequence numbers counter. This is shown on Figure 5.

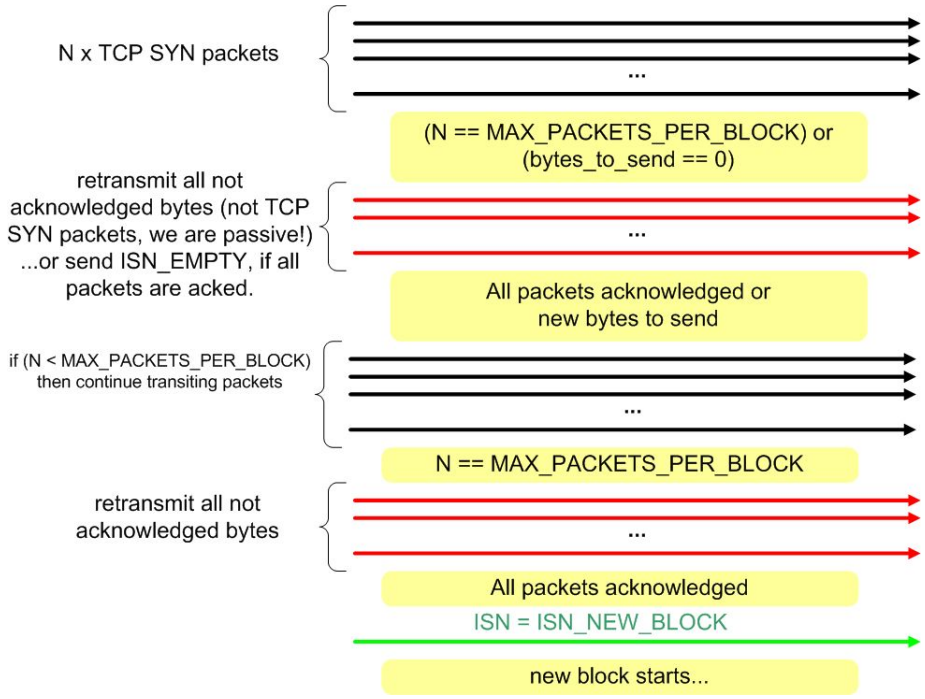

Figure 5. The (simplified) overview of the protocol.

The protocol can be also depicted with the simple diagram, as presented on Figure 6. We can see, that there are two special packets (see Figure 4), which indicate the beginning/end of block and also a special packet telling there were no data sent in the packet. Actually the last packet could be just a special case of the normal data packet, with the number of data byte set to zero. However, some randomness has been added, to make sure that when a block cipher will be used to encrypt the SEQ (see Encryption) it will result in different ISN values sent to the wire (after all we expect many ISN_EMPTY packets)[2].

[2] Actually the encoding algorithm used in the current implementation will generate different results even for the same ISN packets (because it also uses other fields in the packets to generate "one-time" XOR key). However, we could also consider to use different algorithm in the future, so it is probably a good idea to have such randomness in those special packets.

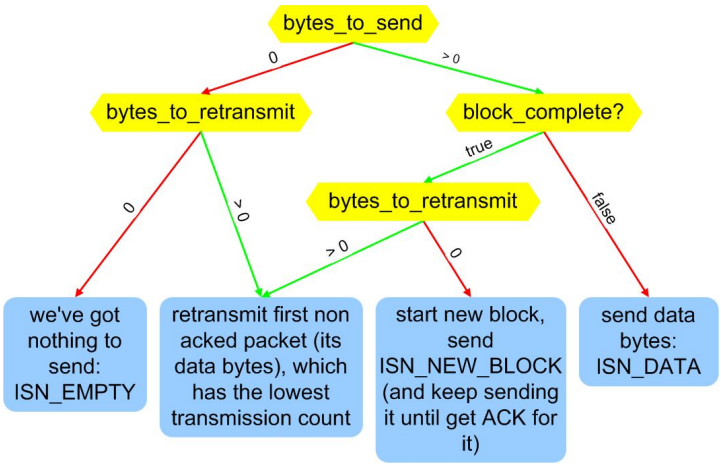

Figure 6. NUSHU protocol diagram.

Encryption

Before sending the ISN (or any other SEQ field) as prepared by the reliability layer (see earlier) we need to encrypt it, mainly because the ISN should look like a random number and not just like plain data, which would allow easy detection of covert channel. The approach taken in the NUSHU implementation is depicted on Figure 7.

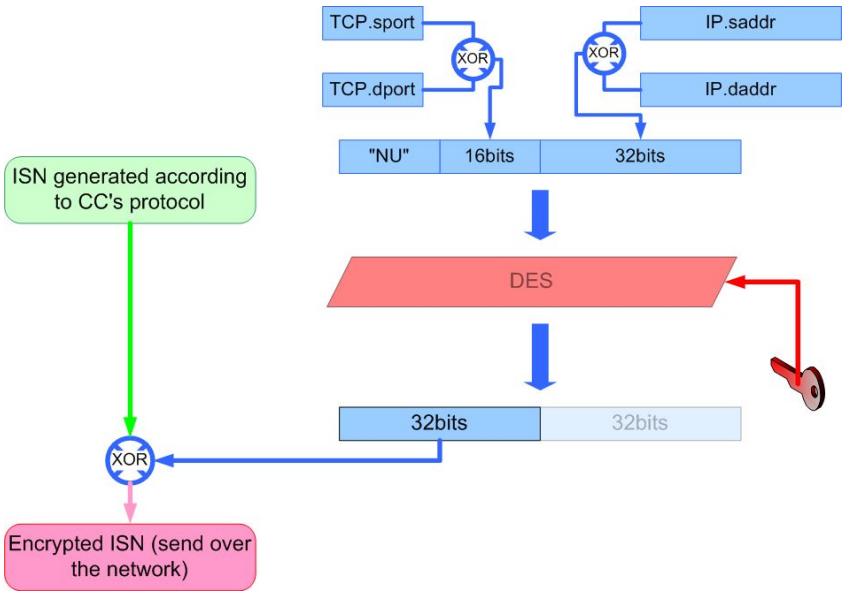

Figure 7. The encryption algorithm for ISN numbers.

Although it may seem quite complex at first glance, it is rather straightforward solution. We cannot use the block cipher directly to encrypt the generated ISN number, because we have only 32 bits to store the result[3] (i.e. the cipher text). Because of this limitation, we first generate an "one-time-pad" XOR key and then XOR the ISN with the first 32bits of the key generated this way. The key is generated with use of some block cipher (DES in current implementation) to ensure its good randomness.

[3] And most good ciphers (that is generating random looking output) operates on blocks not less then 64bits wide.

The input for the block cipher is obviously taken from fields in the TCP packet which carries the encrypted ISN. It may seem that it does not matter which fields we actually use, because receiver will still have access to all of them and will be able to generate XOR key and decrypt the ISN. However, we need to remember that the ISN decryption needs to be done also on the sending side, that is when the ACK packet comes in and the receiver needs to decrypt its ACK to get know which data that packet acknowledges (needs to get access to PCC sequence number). This is the reason that we can only use the following fields in the XOR key generation (these are the only ones, which we can be sure that will be present in the corresponding ACK packet):

- IP source and destination address
- TCP source and destination port

If we also notice that the meaning of "source" and "destination" is reversed in the ACK packet, then the algorithm presented at Figure 7 should become obvious.

It should be noted here, that the security of the cipher plays rather a second role. The most important thing is how similar are the "random numbers" generated by NUSHU to the random ISNs generated by the OS kernel. This have not been verified yet.

"Reverse mode" of TCP ISN channel.

On the Figure 8 we can see a secret channel in the TCP ISN ACK numbers, which could be used against compromised servers.

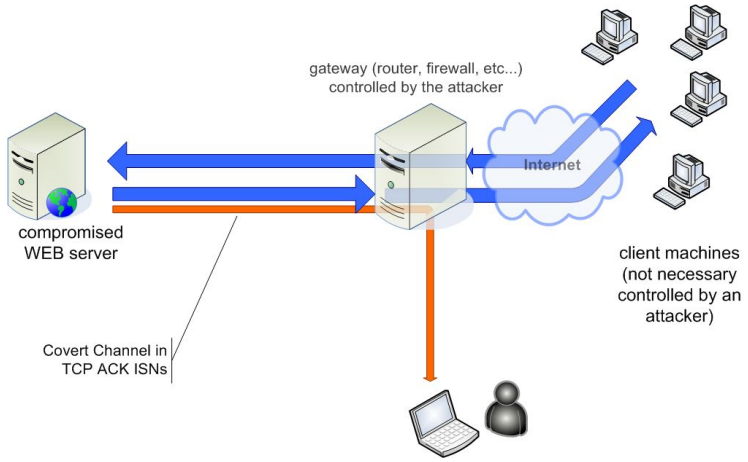

Figure 8. "Reverse mode".

Avoiding local detection (via PF_PACKET sockets)

The PCC operates on very low level, i.e. it changes the ISN field just before the packet is sent over the network and just after it is received by the network interface card. This latter can lead to an easy local detection.

On the `tcpdump` output presented below, taken on the compromised host, we can see a mismatch between the SYN number sent (SEQ field in the first packet) and the ACK number in the second packet. We see the different numbers, because, we actually see the result of the PCC module translations in both cases. In the first packet, we see the number which was inserted by PCC, which contains the secret data which are going to be sent over the network.

In the second packet, we see the result of reverting the ACK field in the incoming packet to the original number, which could be understand by the OS kernel.

```
172.16.100.2.1092 > 172.16.100.1.888: SYN
4500 003c 03ac 4000 4006 16ec ac10 6402
ac10 6401 0444 0378 4242 4242 0000 0000
a002 16d0 7b99 0000 0204 05b4 0402 080a
0018 0921 0000 0000 0103 0300


172.16.100.1.888 > 172.16.100.2.1092: SYN|ACK
4500 003c 0000 4000 4006 1a98 ac10 6401
ac10 6402 0378 0444 1636 5a84 37bf 0a8e ← ACK#
a012 16a0 1e82 0000 0204 05b4 0402 080a
0017 2e9d 0018 0921 0103 0300
```

It should be noted here, that `tcpdump` is actually using PF_PACKET sockets to sniff the traffic. PF_PACKET sockets are actually implemented with the help of `packet_rcv()` kernel function, which in turn is also placed on the `ptype_all` list, just next to our PCC handler.

Surprisingly, if the PF_PACKET handler (i.e. `packet_rcv()`) was put before the PCC handler on that `ptype_all` list, we would still observe the altered SEQ/ACK numbers in the `tcpdump`. This somewhat strange behavior could be explained when we dive into the `dev_queue_xmit_nit()` and `netif_receive_skb()` functions, which are responsible for executing `ptype_all` handlers for outgoing and incoming packets respectively. In both cases all the registered *ptype* handlers operate on the same `skb->data` structure and all the handlers are actually executed in the kernel mode, before the userland PF_PACKET code is executed, which explains the insensitivity for the order of handlers on the `ptype_all` list.

Such behavior is very convenient if outgoing packets are considered, since we do want that the `tcpdump` displays the altered ISN numbers, i.e. the same numbers which are transmitted over the wire. There is a problem, however, with incoming packets, since, in this case, `tcpdump` displays the ACK numbers after PCC reverts them to the original ones (i.e. the ones which were generated by the kernel when SYN packets were assembled), which easy betrays that something storage happens and in fact demystifies the PCC itself.

To solve that problem PCC should redirect all existing *ptype* handlers through an extra function, which will take care of making a copy of the `skb` buffer:

```
int cc_packet_rcv (
 struct sk_buff *skb,
 struct net_device *dev,
 struct packet_type *pt) {

  struct pt_hook_info *pthi = (struct pt_hook_info*) pt->data;
  pt->data = pthi->orig_data;
  int ret;

  if (skb->pkt_type == PACKET_HOST && pt->data != (void*)0xbabe) {
    struct sk_buff * skb2 = skb_copy (skb, GFP_ATOMIC);
    kfree_skb (skb);
    ret = pthi->orig_func (skb2, dev, pt);
  }
```

```
  else ret = pthi->orig_func (skb, dev, pt);

  pthi->orig_data = pt->data;
  pt->data = (void*) pthi;
  return ret;
}
```

To redirect all existing handlers, the PCC module needs to traverse the `ptype_all` list and replace all `ptype->func` pointer as well as store the original pointer in the block of data pointed by `ptype->data`:

```
void hook_ptype (struct packet_type *pt) {
  struct pt_hook_info *pthi = kmalloc (
      sizeof (struct pt_hook_info), GFP_KERNEL);
  pthi->orig_func = pt->func;
  pthi->orig_data = pt->data;
  pt->func = cc_packet_rcv;
  pt->data = (void*)pthi;
}
```

PCC should also take care about possible *ptype* handlers which might be added in the future (because somebody will start `tcpdump` in a later time, for example). To achieve this, `dev_add_pack()` function needs to be hooked [8] and the new function should take care of calling `hook_ptype()` before the original `dev_add_pack()` gets executed.

NUSHU – sample implementation

NUSHU [9] is the sample implementation of TCP ISN based passive covert channel for Linux kernels. It consists of two main modules:

- `nushu_sender.o`
- `nushu_receiver.o`

which should be loaded on the compromised host and on the attacker's controlled gateway (see Figure 1).

This sample implementation should be considered as proof-of-concept code, since it is only the communication channel engine. For practical use it is necessary to combine it with some host based information sniffer, like password key logger, which will exploit the channel for leaking the sniffed information.

NUSHU provides however some advanced features, which include:

- Reliability layer implemented as a simple protocol (with passive receiver), which takes care of packet reordering and lost packet retransmissions if necessary (see Reliability layer).
- Encryption of all generated ISN numbers, to make NUSHU detection hard, even for people using statistical analysis (see Encryption).
- Local PF_PACKET sockets cheating module to be used on the compromised host as described in Avoiding local detection (via PF_PACKET sockets) above. This is implemented in `nushu_hider.o` module.

Future work

Feature work on the passive covert channels may consider the following tasks:

- Porting NUSHU to another operating systems. Windows XP seems to be the first candidate for this.
- Implementing bidirectional channel, which could be used not only for leaking information from the compromised host, but also for controlling it in a backdoor-like manner.
- Network based detector, which will analyze the randomness of the ISN numbers in the TCP flows. Although NUSHU use DES for making the generated ISN numbers look random, we can expect that some characteristics will be different comparing to the characteristics of the random numbers generated by the operating system.
- Testing another couriers then TCP ISN, like HTTP Cookies for example, which could have the advantage of bypassing the proxy servers.

Credits

All members of `#convers` channel for interesting conversations :)

References

[1] Linux Kernel Sources, `http://kernel.org`

[2] Netfilter Official Documentation, `http://netfilter.org`

[3] kossak, *Building Into The Linux Network Layer*, Phrack Magazine, Issue 55, `http://phrack.org`

[4] bioforge, *Hacking the Linux Kernel Network Stack*, Phrack Magazine, Issue 61, `http://phrack.org`

[5] Craig H. Rowland, *Covert Channels in the TCP/IP Protocol Suite*, First Monday, 1996, `http://www.firstmonday.dk/issues/issue2_5/rowland/`.

[6] John Giffin et al., *Covert Messaging Through TCP Timestamps*, Massachusetts Institute of Technology, 2002.

[7] Andrew Hintz, Covert Channels in TCP and IP Headers, 2003, `http://guh.nu/projects/cc/covertchan.ppt`.

[8] mayhem, *Linux x86 kernel function hooking emulation*, Phrack Magazine, Issue 51, `http://phrack.org`.

[9] Joanna Rutkowska, *NUSHU*, sample implementation of unidirectional passive covert channel in the TCP ISN numbers, 2004, `http://invisiblethings.org/tools.html`.