Passive Covert Channels Implementation in Linux Kernel

Joanna Rutkowska
joanna (at) invisiblethings (dot) org

Chaos Communication Congress,
December 27th -29th 2004, Berlin
Passive Covert Channels

- Do not generate their own traffic
- Only change some fields in the packets generated by user (like HTTP requests)
- Best used for stealing data from desktop computers
- Usually requires that the attacker control the company’s gateway (for example works in ISP)
- Typical usage: information stealing from corporate Workstations and servers (in “mirror mode”, see later)
Passive Covert Channels

- **Compromised computer**
- **Covert Channel** (steganography in TCP/IP headers for example)
- **Gateway (router, firewall, etc...)** controlled by the attacker
- Internet
- www.google.com
How to implement?

Let’s first have a look at how packets are handled inside the Linux kernel…
Handling Incoming Packets

- **Interrupt Service Routine**
  - `enqueue(dev)`
  - `netif_rx_schedule`
    - `raise_softirq`
      - `NET_RX_SOFTIRQ`

- **`sk_buff`**
- **`net_device`**
- **`dev_poll`**
  - Receives packet and creates `sk_buff` for it.
  - `netif_receive_skb`
    - Decision based on `skb->protocol` field.
  - `arp_rcv()`, `ip_rcv()`, `packet_rcv()`

- `dev = dequeue()`
  - `net_rx_action`
  - `dev->poll()`

- `netif_rx`
  - (obsolete drivers, non-NAPI)
  - `CPU1`, `CPU2`

- Apply to the new NAPI architecture (kernels >= 2.4.20 & 2.6.x)
Incoming IP packets

initial verification of IP packets

Netfilter makes the decision:
1) NF_DROP
2) NF_ACCEPT
3) NF_STOLEN
...

skb->dst->input = ?

Set skb->dst field

ip_local_delivery  ip_forward  ip_mr_input  ip_err

MULTICAST forwarding
Local delivery

```
<table>
<thead>
<tr>
<th></th>
<th>ip_local_deliver</th>
<th>ip_defrag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NF_IP_LOCAL_IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ip_local_deliver_finish</td>
<td></td>
</tr>
<tr>
<td>skb-&gt;nh.iph-&gt;protocol = ?</td>
<td>raw_v4_input</td>
<td>tcp_v4_rcv</td>
</tr>
<tr>
<td></td>
<td>udp_rcv</td>
<td>icmp_rcv</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

Forwarding packets

- ip_forward
  - NF_IP_FORWARD
    - ip_forward_finish
      - ip_output
        - ip_finish_output
          - NF_IP_POST_ROUTING
            - ip_finish_output2
              - Send the packet...

- Decrease TTL, send ICMP Error message if necessary, etc...
- If there are any options to process in IP header
- Packets sent from local processes (sockets) (TCP, UDP, etc...)

Outgoing packets

- `ip_finish_output2`:
  - `dev_queue_xmit`
    - enqueue skb in the device queue
  - `dev->qdisc`
    - `qdisc_restart`
    - `dev->hard_start_xmit`
    - Send the packet...
  - `dev_queue_xmit_nit`
    - call all the packet type handlers registered on `ptype_all_list`

Two important techniques

ptype_* handlers

Netfilter hooks
Protocol handlers

ptype_all

ETH_IP_ALL

next prev

ip_rcv

packet_rcv

arp_rcv

ipv6_rcv

ETH_P_IP

next prev

ETH_P_ARP

next prev

ETH_P_IPV6

next prev

ptype_base

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev

next prev
Key structure: packet_type

```c
struct packet_type
{
    unsigned short type;  // htons(ether_type)
    struct net_device *dev;  // NULL means all dev
    int (*func)(...);  // handler address
    void *data;  // private data
    struct list_head list;
};
```

There are two exported kernel functions for adding and removing handlers:

- `void dev_add_pack(struct packet_type *pt)`
- `void dev_remove_pack(struct packet_type *pt)`
Addition of own handler

```c
struct packet_type myproto;

myproto.type = htons(ETH_P_ALL);
myproto.func = myfunc;
myproto.dev = NULL;
myproto.data = NULL;

dev_add_pack (&myproto)
```
Passive Covert Channels

- Compromised computer
- Gateway (router, firewall, etc...) controlled by the attacker
- Covert Channel (steganography in TCP/IP headers for example)

www.google.com
Internet

The SEQ#, which is transited first is called Initial Sequence Number (ISN)
TCP handshake

SYN: SEQ = ISN₁, ACK = 0

SYN|ACK: SEQ = ISN₂, ACK = ISN₁+1

ACK: SEQ = ISN₁+1, ACK = ISN₂+1
Idea of ISN based passive CC

- Change ISN numbers in all (or only some) outgoing TCP connections (on compromised host)
- Make sure to change back the ACK numbers in incoming connections, co kernel will not discard the packets
- Also, change SEQ# in all consecutive packets belonging to the same TCP connection
- We can send 4 bytes per TCP connection this way.
- Not much, but when considering lots of HTTP connections made by ordinary users it should be ok for sending for e.g. sniffed passwords, etc...
Passive TCP ISN covert channel idea

Linux Kernel

secure_tcp_sequence() → ISN_{orig} → TCP SYN

Covert Channel Kernel Module

CC's generated ISN → ISN_{CC} → TCP SYN

- (ISN_{CC} - ISN_{orig})

TCP SYN|ACK → TCP SYN|ACK

+ (ISN_{CC} - ISN_{orig})

TCP ACK → TCP ACK

network

TCP SYN|ACK → TCP SYN|ACK

TCP ACK → TCP ACK
Tracing TCP connections

For each TCP connection a block of data is allocated (by a CC kernel module):

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

It allows you to correctly change the SEQ numbers of all incoming and outgoing TCP packets
Detecting end of connection

- After the user close the connection it would be nice that the CC module free the conn_info structure for that connection (memory in kernel is a an important resource)
- We can implement TCP state machine in CC module to detect when the connection is actually closed (and we don’t need to worry about changing its SEQ/ACK numbers anymore)
  - but this is too much work;;)
- Another solution: look at the kernel tcphash_info, which holds all information about live TCP connections
- From time to time remove dead TCP connection info (struct conn_info).
Adding Reliability Layer

- Any communication channel without reliability mechanism is not really useful outside lab.
- In ISN based CC we can exploit the nature of TCP protocol: every SYN packet is acknowledged either by SYN|ACK or by RST packet.
- All we need to do is to trace which packets were actually acknowledged.
- We need to add packet ordering (our own sequence numbers).
Protocol

N x TCP SYN packets

(retransmit all not acknowledged bytes (not TCP SYN packets, we are passive!)
...or send ISN_EMTPY, if all packets are acked.

if (N < MAX_PACKETS_PER_BLOCK)
then continue transiting packets

(retransmit all not acknowledged bytes)

(N == MAX_PACKETS_PER_BLOCK) or (bytes_to_send == 0)

All packets acknowledged or new bytes to send

All packets acknowledged

ISN = ISN_NEW_BLOCK

new block starts...

Note that receiver is passive!
Protocol Diagram

- **bytes_to_send**
  - 0
  - > 0

- **block_complete?**
  - true
  - false

- **bytes_to_retransmit**
  - 0
  - < 0
  - > 0
  - 0

- **we've got nothing to send:**
  - ISN_EMPTY

- **retransmit first non acked packet (its data bytes), which has the lowest transmission count:**

- **start new block, send ISN_NEW_BLOCK (and keep sending it until get ACK for it):**

- **send data bytes:**
  - ISN_DATA

Protocol implementation: TCP ISN field

<table>
<thead>
<tr>
<th>control byte</th>
<th>data byte #2</th>
<th>data byte #1</th>
<th>data byte #0</th>
</tr>
</thead>
</table>

This is the 32bit SEQ (or ACK) field from TCP packet

# of actual data bytes sent in this packet:
00: no data (control packet)
01: b0 is valid
10: b0 & b1 are valid
11: b0, b1 & b2 are valid
## Special packets

<table>
<thead>
<tr>
<th>ISN_NEW_BLOCK</th>
<th>ISN_EMPTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 tcp.sport 0xAA</td>
<td>0x00 tcp.sport 0xFF</td>
</tr>
</tbody>
</table>

Special packets contain “random” bytes, to avoid duplicated ISN numbers (which could easily betray the covert channel). Remember that all ISN’s are encrypted with a block cipher before sending to the wire.
ISN Encryption

✦ Every ISN, generated by CC protocol engine, is encrypted with a block cipher (see later)
✦ Both sides share the common key
✦ Probably the most important thing about the algorithm used is how similar the characteristics of the "random" numbers it generates are to the ISN numbers generated by the Linux kernel.
✦ The security of the cipher algorithm plays rather second role here, since it seems unlikely that anybody will try to break it;)

ISN Encryption

- ISN (SEQ) field is only 32 bit wide
- Most good block cipher operates on blocks greater or equal to 64 bits
- Solution: Use DES to generate a “one-time-pad” key and xor ISN with the lowest 32bits of the generated key.
- We use TCP source and destination port and IP source and destination address as a “seed” to generate key.
ISN Encryption

ISN generated according to CC's protocol

TCP.sport

TCP.dport

IP.saddr

IP.daddr

"NU" 16bits 32bits

DES

32bits 32bits

Encrypted ISN (send over the network)
Encryption

- NOTE: we can only use these elements to generate key, since we need to assure that not only the receiver will be able to decrypt it but also the sender, when decrypting the ACK packet’s ACK#!
- This is also the reason for XORing destination and source, so we don’t need to worry about reversing them when considering the ACK packet.

SYN: SEQ = ISN₁, ACK = 0

We need to be able to decrypt this number too!

SYN|ACK: SEQ = ISN₂, ACK = ISN₁+1
Nü Shu __

- Secret language of Chinese women
- Characters were often disguised as decorative marks or as part of artwork
- Existed for centuries, but was not known to most of the world until 1983!
NUSHU – TCP ISN based passive Covert Channel

Features:
- on-the-fly SEQ# changing
- Reliability layer
- PF_PACKET cheating
- For Linux 2.4 kernels (see later discussion on 2.6 kernels)
Time to show some working code :)
PART II

- Inquisitive PF_PACKETS
- Cheating local PF_PACKETs sniffers + DEMO
- “Reverse mode” & bidirectional channels
- Host based detection + DEMO
- Discussion of network based detection
- Some notes about hiding LKMS and LKMs in 2.6 kernels
Inquisitive PF_PACKET sockets

Q: If you try running `tcpdump` on a host compromised with NUSHU, what will happen?

A: The outgoing packets will have the ISN displayed correctly (i.e. the ISN inserted by CC). However, the incoming TCP packets will have the ISN displayed incorrectly (i.e. the ISN after the CC changed it)

Surprisingly, this behavior doesn’t depend on whether the PF_PACKET socket (the `tcpdump`’s one) was loaded before or after the CC module registered its handler!
local tcpdump problem

[SYN packet as seen on compromised host (172.16.100.2)]:
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

[SYN|ACK packet, again, as seen on compromised host]:
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
  a012 16a0 1e82 0000 0204 05b4 0402 080a
  0017 2e9d 0018 0921 0103 0300

ACK# (should be: 0x43424242)
**skb_clone() vs skb_copy()**

```c
dev_queue_xmit_nit (skb, ...) {
    skb2 = skb_clone(skb);
    (...)
    ptype->func (skb2);
}
```

- Every ptype handler operates de facto on the same data (`skb->data` is not copied during `skb_clone()`).
- If the CC’s ptype handler is called before `PF_PACKETS’s packet_rcv()`, then `tcpdump` displays the changed SEQ#.
- When the `packet_rcv()` is called first, the userland process’ socket still gets the wrong data, since it effectively reads the data (`skb->data`) after all the kernel stuff is executed on this packet.
PF_PACKET Cheating idea

- Redirect all ptype handlers calls, except CC’s one, through additional function \( \text{cc	extunderscore packet	extunderscore rcv} \), which will copy (not clone!) the skb buffer and call original handler.

- To do this:
  - Traverse \texttt{ptype	extunderscore all list} and replace all \( \text{pt	extunderscore func} \) to point to \texttt{cc	extunderscore packet	extunderscore rcv()}
  - hook \texttt{dev	extunderscore add	extunderscore pack() } to catch all future ptype registrations
int cc_packet_rcv (struct sk_buff skb, ...) {
    skb2 = skb_copy (skb);
    if (incoming_packet and
        orig_func != cc_func)
        return orig_func (skb2, ...);
    else return orig_func (skb, ...);
}

void cc_dev_add_pack (pt) {
    pt->func = cc_packet_rcv;
    pt->data = {orig_func, orig_data};
}
Live DEMO

NUSHU + PF_PACKET cheating
Server mode ("reverse mode")

This time covert channel is in ACK# fields, not in #SEQ.
Bidirectional channels

In this presentation we focused on information stealing, rather than backdoor technology (thus unidirectional channels)

NUSHU could pretty easily be extended to support bidirectional transmission:

- one direction: SYN packet’s ISN
- opposite direction: SYN ACK packet’s ISN

SYN: SEQ = ISN₁, ACK = 0

SYN|ACK: SEQ = ISN₂, ACK = ISN₁+1
Covert Channels Detection

Host Based

Network Based
Detecting extra ptype handler (host based detection)

- invasive (requires a special module, which registers a dummy ptype handler for a while)
- noninvasive (does not require any kernel changes, can be implemented through /dev/kmem)
How to detect?

- How to get a list of registered protocol handlers?
- Author does not know any tool (or even kernel API) for doing that!
- We need to “manually” check the following lists:
  - ptype_all
  - ptype_base
- But their addresses are not exported!
Where are the protocol lists?

Two kernel global variables (net/core/dev.c):
- `static struct packet_type *ptype_base[16];`
- `static struct packet_type *ptype_all = NULL;`

Only the following functions are referencing those variables (i.e. “know” their addresses):

<table>
<thead>
<tr>
<th>Kernel 2.4.20</th>
<th>Kernel 2.6.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. dev_add_pack()</td>
<td>1. dev_add_pack()</td>
</tr>
<tr>
<td>2. dev_remove_pack()</td>
<td>2. __dev_remove_pack()</td>
</tr>
<tr>
<td>3. dev_queue_xmit_nit()</td>
<td>3. dev_queue_xmit_nit()</td>
</tr>
<tr>
<td>4. netif_receive_skb()</td>
<td>4. netif_receive_skb()</td>
</tr>
<tr>
<td>5. net_dev_init()</td>
<td></td>
</tr>
</tbody>
</table>

The functions in **green** are exported.
Approaches for finding the lists

- **System.map file**
  - problem: the file is not always up to date or sometimes it does not even exist (for security reasons;))

- **“heuristic” method**
  - We know the addresses of several functions which are using the addresses we are looking for.
  - We can look at their body to find all the 32 bit words which look like kernel pointers.
  - We then need to find the common set of those pointer-like words from all functions we considered.
  - Finally we need to check every potential value from the common subset to see if it looks like (or could be) the ptype_all or ptype_base list head.
Illustration for the heuristic method

Potential addresses obtained from
function `dev_add_pack()`

Potential addresses obtained from
function `__dev_remove_pack()`

ptype_all real address should be one of the addresses from the common subset (in practice it contains about 3-4 addresses)
Live DEMO: detecting additional protocol handlers

PTYPE_ALL:
hook type ETH_P_ALL (0x3)
hook at: 0xc487e060  [module: unknown module]

PTYPE_BASE[]:
hook type ETH_P_IP (0x800)
hook at: 0xc0203434 -> ip_rcv() [k_core]

hook type ETH_P_802_2 (0x4)
hook at: 0xc01f8050  [k_core]

hook type ETH_P_ARP (0x806)
hook at: 0xc0223778 -> arp_rcv() [k_core]
“Invasive” method

Write a little module, which adds its own (dummy) packet type handlers:

```c
int dummy_handler (...) { return 0; }
myproto.type = ETH_P_ALL;
myproto.func = dummy_handler;
dev_add_pack (&myproto);
```

So, you can now traverse the interesting list, starting from: `myproto.next`

After reading all the handler addresses, you can simple deregister the dummy protocol handler.
We mentioned only two possible ways of implementing passive covert channels:
- ptype handlers
- Netfilter hooks

These are the easiest (and probably most elegant).

But there are many other possible ways to create covert channels in the Linux kernel, for example:
- internal kernel function hooking (biggest problem: most of them are not exported). Quite easy to detect.
- function pointer hooking, like:
  - `arp_*_ops.hh_output`
  - `net_device.poll`
  - etc...

...hard to detect!
host-based backdoor and covert channel detector

Properly implemented host-based compromise detector, should:
• Checks for hidden processes
• Checks for hidden sockets
• Checks ptype handlers (noninvasive method)
• Checks Netfilter hooks
• Checks integrity of kernel code (ala Tripwire)
• Checks important network code pointers
Network Based Detection

- The characteristics of ISN numbers generated by NUSHU will be different from the ISN generated by Linux Kernel.
- We need a reliable method for fingerprinting PRNG.
- We have to save the correct PRNG (Linux kernel’s) characteristics in a detector database.
- The detector measures the characteristics of the suspected TCP flows and compares them against the stored fingerprints (note: detector must be told which exact OSs are running in the network).
- Writing a network based covert channel detector is on my TODO list ;)

Notes about stealth modules

- load module as usual (insmod)
- in init_module():
  - allocate some memory by kalloc()
  - do not use vmalloc(), since such memory goes beyond (phys_mem + VMALLOC_START), which makes it easy to detect
  - copy all code and global data to allocated buffer
  - relocate code
- remove module (rmmod)
- NOTE: /dev/kmem cannot be used on for example Fedora Core 2&3 systems.
Linux 2.6 Considerations

- Changed module loading scheme:
  http://lwn.net/Articles/driver-porting/

- There is no compatibility at binary level for modules anymore (no MODVERSIONS)! :-o

- Each module needs to be recompiled for the exact kernel version
  You can expect some strange incompatibility issues, like different structure layouts between one minor kernel version to another (for example struct module has been changed in 2.6.6, breaking all binary compatibility)

- Besides that, seems to be no important differences which would make the implementation difficult
Linux 2.6 LKM hell

- Special macro, VERMAGIC_STRING, has been added to allow checking if the module matches the kernel
- When trying to load test.ko module built for Fedora Core 2 on a Slackware 10 system we get the following error (vermagic mismatch):
  
  test: version magic '2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3' should be '2.6.7 486 gcc-3.3'

- We see a calling convention mismatch and different stack sizes. Loading such module will probably crash the system
# VERMAGIC_STRING

**include/linux/vermagic.h:**

```c
#define VERMAGIC_STRING \
    UTS_RELEASE " " \ // e.g: "2.6.5-1.358"
MODULE_VERMAGIC_SMP \ // “SMP” or “”
MODULE_VERMAGIC_PREEMPT \ // “preempt” or “”
MODULE_ARCH_VERMAGIC \ // see below
"gcc-" __stringify(__GNUC__) "." \ 
__stringify(__GNUC_MINOR__) // “gcc-3.3”
```

**include/asm-i386/module.h:**

```c
#define MODULE_ARCH_VERMAGIC \ 
    MODULE_PROC_FAMILY \ // e.g. “PENTIUM4”
MODULE_REGPARM \ // “REGPARM” or “”
MODULE_STACKSIZE \ // “4KSTACKS” or “”
```
Future work

- Windows port
- Bidirectional channel
- Network based detector (statistical analysis, PRNG fingerprinting)
- Different courier then TCP ISN (HTTP Cookie?)
Credits & Greets

- All members of the #convers channel
- Ian Melven
- Paul Wouters
- JG