Code Monkey Tips

Code development and system administration tips for which I did not find a solution online.

Thursday, July 14, 2011

Zero-copy network transmission with vmsplice

Download the complete sourcecode as zerocopy-vmsplice.c

This post completes a set that also includes asynchronous reading with PACKET_RX_RING and asynchronous writing with PACKET_TX_RING. In this post, we look at sending packets out over a raw socket in zero-copy fashion.

First, understand that the presented code is a **silly hack**: it requires four system calls for each transmitted packet, so will never be fast. I post it here because the code can be helpful in other situations where you want to use vmsplice.

Second, note that splice support is a protocol-specific feature. Raw sockets support it as of recent kernels. I believe 2.6.36 has it, but would not be surprised if it is lacking in 2.6.34, for instance (please leave a comment if you know when it was introduced).

The basic idea is to send data to a network socket without copying using vmsplice(). But, the vmsplice syscall will only splice into a pipe, not a network socket. Thus, the data first has to be appended to a pipe and then has to be moved to the socket using a splice() call. One extra complication is that vmsplice() works on entire pages (as it relies on memory protection mechanisms). In this example, we transport a single packet per page, which mean that we have to flush the rest of the page contents to /dev/null. It is not impossible to fill a page with multiple packets and then splice() them to the network -- this indeed sounds much more worthwhile.

On to the code. I lifted this code from another project which always stored one packet per page, not necessarily page-aligned. That is most definitely not a requirement of splicing. In general, try to align pack multiple packets in a page and have the first be page aligned.

```
/// transmit a packet using splice
static int
do_transmit(void *page, int pkt_offset, int pktlen)
{
  struct iovec iov[1];
  int ret, len_tail;

  // send page to kernel pipe
  iov[0].iov_base = page;
  iov[0].iov_len = getpagesize();

  ret = vmsplice(tx_splicefd[1], iov, 1, SPLICE_F_GIFT);
  if (ret != getpagesize()) {
    fprintf(stderr, "vmsplice()\n");
    return 1;
  }

  // splice unused headspace to /dev/null (because our packet is not aligned)
  ret = splice(tx_splicefd[0], NULL, tx_nullfd, NULL, pkt_offset, SPLICE_F_MOVE);
  if (ret != pkt_offset) {
    fprintf(stderr, "splice() header\n");
    return 1;
```

I took the red pill

Blog Archive

About Me

Willem

View my complete profile

```
  }

  // splice or sendfile packet to tx socket
  ret = splice(tx_splicefd[0], NULL, tx_rawsockfd, NULL, pktlen, SPLICE_F_MOVE);
  if (ret != pktlen) {
    fprintf(stderr, "splice() main\n");
    return 1;
  }

  // splice unused tailspace to /dev/null
  len_tail = getpagesize() - pktlen - pkt_offset;
  ret = splice(tx_splicefd[0], NULL, tx_nullfd, NULL, len_tail, SPLICE_F_MOVE);
  if (ret != len_tail) {
    fprintf(stderr, "splice() footer\n");
    return 1;
  }
  return 0;
}
```

This code makes use of one pipe and two other file descriptors. tx_splicefd is a regular pipe, tx_nullfd is an open file handle to /dev/null and tx_rawsockfd is a raw IP socket. They were created as follows:

```
/// source IP address in host byte order
#define CONF_TXHOST_HBO ((127 << 1)) + 1

static int tx_splicefd[2], tx_nullfd, tx_rawsockfd;

/// open a RAW or UDP socket for retransmission
//  @return 0 on success, -1 on failure
static int
do_init(void)
{
  struct sockaddr_in saddr;

  // open tx socket
  tx_rawsockfd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
  if (tx_rawsockfd < 0) {
    perror("socket() tx");
    return -1;
  }

  // configure raw socket
  memset(&saddr, 0, sizeof(saddr));
  saddr.sin_family      = AF_INET;
  saddr.sin_port        = htons(ETH_P_IP);
  saddr.sin_addr.s_addr = htonl(CONF_TXHOST_HBO);

  if (connect(tx_rawsockfd, &saddr, sizeof(saddr))) {
    perror("connect() tx");
    return -1;
  }

  // when splicing, have to first send to kernel pipe, then to tx socket.
  // also, unwanted data must be flushed to /dev/null
  // create pipe to splice to kernel
  if (pipe(tx_splicefd)) {
    perror("pipe() tx");
    return -1;
  }

  // open /dev/null for splicing trash
  tx_nullfd = open("/dev/null", O_WRONLY);
  if (tx_nullfd < 0) {
    perror("open() /dev/null");
    return -1;
  }
```

```
  return 0;
}
```

Posted by Willem at 2:45 PM 1 comment:

Labels: bsd sockets, splice, vmsplice, zero-copy

Asynchronous packet socket writing with PACKET_TX_RING

Download the complete sourcecode as packet-tx-ring.c

In my last post, I showed how you can read packets enqueued on a packet socket without system calls, by setting up a memory mapped ring buffer between kernel and userspace. Since version 2.6.31, the kernel also supports a transmission ring (or at least, the macro exists since that version; I tested this code against version 2.6.36).

Setting up of a transmission ring is trivial once you know how to create a reception ring. In the setup snippet of the previous post, simply change the call to init_packet_sock to read

```
  fd = init_packetsock(&ring, PACKET_TX_RING);
```

Then, at runtime, write packets as follows:

```
/// transmit a packet using packet ring
//  NOTE: for high rate processing try to batch system calls,
//        by writing multiple packets to the ring before calling send()
//
//  @param pkt is a packet from the network layer up (e.g., IP)
//  @return 0 on success, -1 on failure
static int
process_tx(int fd, char *ring, const char *pkt, int pktlen)
{
  static int ring_offset = 0;

  struct tpacket_hdr *header;
  struct pollfd pollset;
  char *off;
  int ret;

  // fetch a frame
  // like in the PACKET_RX_RING case, we define frames to be a page long,
  // including their header. This explains the use of getpagesize().
  header = (void *) ring + (ring_offset * getpagesize());
  assert((((unsigned long) header) & (getpagesize() - 1)) == 0);
  while (header->tp_status != TP_STATUS_AVAILABLE) {

    // if none available: wait on more data
    pollset.fd = fd;
    pollset.events = POLLOUT;
    pollset.revents = 0;
    ret = poll(&pollset, 1, 1000 /* don't hang */);
    if (ret < 0) {
      if (errno != EINTR) {
        perror("poll");
        return -1;
      }
      return 0;
    }
  }

  // fill data
  off = ((void *) header) + (TPACKET_HDRLEN - sizeof(struct sockaddr_ll));
```

```
  memcpy(off, pkt, pktlen);

  // fill header
  header->tp_len = pktlen;
  header->tp_status = TP_STATUS_SEND_REQUEST;

  // increase consumer ring pointer
  ring_offset = (ring_offset + 1) & (CONF_RING_FRAMES - 1);

  // notify kernel
  if (sendto(fd, NULL, 0, 0, (void *) &txring_daddr, sizeof(txring_daddr)) < 0) {
    perror("sendto");
    return -1;
  }

  return 0;
}
```

As the function comment says, this example makes inefficient use of the ring, because it issues a send() call for every packet that it writes. The whole purpose of the ring is to transmit multiple packet without having to issue a system call (and cause a kernel-mode switch).

The function also makes use of global variable txring_daddr that has not yet been introduced. Packets are copied to the Tx ring from the network layer up. This destination address structure contains the link layer information that the kernel needs to complete the packet. I do not know why we cannot just write packets from the link layer up, but this works. The following snippet sets up a destination address structure. It fills in the destination link layer as ff.ff.ff.ff.ff.ff. Replace this with a sane address in your code.

```
static struct sockaddr_ll txring_daddr;

/// create a linklayer destination address
//  @param ringdev is a link layer device name, such as "eth0"
static int
init_ring_daddr(const char *ringdev)
{
    struct ifreq ifreq;

    // get device index
    strcpy(ifreq.ifr_name, ringdev);
    if (ioctl(fd, SIOCGIFINDEX, &ifreq)) {
      perror("ioctl");
      return -1;
    }

    txring_daddr.sll_family     = AF_PACKET;
    txring_daddr.sll_protocol  = htons(ETH_P_IP);
    txring_daddr.sll_ifindex   = ifreq.ifr_ifindex;

    // set the linklayer destination address
    // NOTE: this should be a real address, not ff.ff....
    txring_daddr.sll_halen     = ETH_ALEN;
    memset(&txring_daddr.sll_addr, 0xff, ETH_ALEN);
}
```

The sockaddr_ll structure is defined in <netpacket/packet.h>

Posted by Willem at 2:17 PM No comments:

Labels: packet socket, PACKET_TX_RING

Asynchronous packet socket reading with PACKET_RX_RING

Download the complete sourcecode as packet-rx-ring.c

Since Linux 2.6.2x, processes can read network packets asynchronously using a packet socket ring buffer. By setting the socket option SOL_SOCKET PACKET_RX_RING on a packet socket, the kernel allocates a ring buffer to hold packets. It will then copy all packets that a caller would have had to read using read() to this ring buffer. The caller then maps the ring into its virtual memory by executing an mmap() call on the packet socket and from then on can read packets without issuing any system calls. It signals the kernel that it has finished processing a packet by setting a value in a header structure that is prefixed to the packet. If the caller has processed all outstanding packets, it can block by isssuing a select() involving the packet socket.

This snippet shows how to set up a packet socket with ring

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>

#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <poll.h>

#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <sys/stat.h>

#include <linux/if_packet.h>

/// The number of frames in the ring
//  This number is not set in stone. Nor are block_size, block_nr or frame_size
#define CONF_RING_FRAMES          128

/// Offset of data from start of frame
#define PKT_OFFSET      (TPACKET_ALIGN(sizeof(struct tpacket_hdr)) + \
                         TPACKET_ALIGN(sizeof(struct sockaddr_ll)))

/// (unimportant) macro for loud failure
#define RETURN_ERROR(lvl, msg) \
  do {                    \
    fprintf(stderr, msg); \
    return lvl;            \
  } while(0);

/// Initialize a packet socket ring buffer
//  @param ringtype is one of PACKET_RX_RING or PACKET_TX_RING
static char *
init_packetsock_ring(int fd, int ringtype)
{
  struct tpacket_req tp;
  char *ring;

  // tell kernel to export data through mmap()ped ring
  tp.tp_block_size = CONF_RING_FRAMES * getpagesize();
  tp.tp_block_nr = 1;
  tp.tp_frame_size = getpagesize();
  tp.tp_frame_nr = CONF_RING_FRAMES;
  if (setsockopt(fd, SOL_PACKET, ringtype, (void*) &tp, sizeof(tp)))
    RETURN_ERROR(NULL, "setsockopt() ring\n");

  // open ring
```

```
  ring = mmap(0, tp.tp_block_size * tp.tp_block_nr,
              PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if (!ring)
    RETURN_ERROR(NULL, "mmap()\n");

  return ring;
}

/// Create a packet socket. If param ring is not NULL, the buffer is mapped
//  @param ring will, if set, point to the mapped ring on return
//  @return the socket fd
static int
init_packetsock(char **ring, int ringtype)
{
  int fd;

  // open packet socket
  fd = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_IP));
  if (fd < 0)
    RETURN_ERROR(-1, "Root priliveges are required\nsocket() rx. \n");

  if (ring) {
    *ring = init_packetsock_ring(fd, ringtype);

    if (!*ring) {
      close(fd);
      return -1;
    }
  }

  return fd;
}

static int
exit_packetsock(int fd, char *ring)
{
  if (munmap(ring, CONF_RING_FRAMES * getpagesize())) {
    perror("munmap");
    return 1;
  }

  if (close(fd)) {
    perror("close");
    return 1;
  }

  return 0;
}

/// Example application that opens a packet socket with rx_ring
int
init_main(int argc, char **argv)
{
  char *ring;
  int fd;

  fd = init_packetsock(&ring, PACKET_RX_RING);
  if (fd < 0)
    return 1;

  // TODO: add processing. See next snippet.

  if (exit_packetsock(fd, ring))
    return 1;

  return 0;
}
```

This snippet shows how to process packets at runtime using the packet ring. The first function reads a single packet from the ring, the second updates the header in the ring to release the frame back to the kernel.

```
static int rxring_offset;

/// Blocking read, returns a single packet (from packet ring)
static void *
process_rx(const int fd, char *rx_ring)
{
  struct tpacket_hdr *header;
  struct pollfd pollset;
  int ret;

  // fetch a frame
  header = (void *) rx_ring + (rxring_offset * getpagesize());
  assert((((unsigned long) header) & (getpagesize() - 1)) == 0);

  // TP_STATUS_USER means that the process owns the packet.
  // When a slot does not have this flag set, the frame is not
  // ready for consumption.
  while (!(header->tp_status & TP_STATUS_USER)) {

    // if none available: wait on more data
    pollset.fd = fd;
    pollset.events = POLLIN;
    pollset.revents = 0;
    ret = poll(&pollset, 1, -1 /* negative means infinite */);
    if (ret < 0) {
      if (errno != EINTR)
        RETURN_ERROR(NULL, "poll()\n");
      return NULL;
    }
  }

  // check data
  if (header->tp_status & TP_STATUS_COPY)
    RETURN_ERROR(NULL, "skipped: incomplete packed\n");
  if (header->tp_status & TP_STATUS_LOSING)
    fprintf(stderr, "dropped packets detected\n");

  // return encapsulated packet
  return ((void *) header) + PKT_OFFSET;
}

// Release the slot back to the kernel
static void
process_rx_release(char *rx_ring)
{
  struct tpacket_hdr *header;

  // clear status to grant to kernel
  header = (void *) rx_ring + (rxring_offset * getpagesize());
  header->tp_status = 0;

  // update consumer pointer
  rxring_offset = (rxring_offset + 1) & (CONF_RING_FRAMES - 1);
}
```

This code was copied from a project that required two separate functions. In most cases, you want to read, process and release a frame in a single loop. I'm not particularly proud of using a global variable for the current ring offset. Download the complete sourcecode as packet-rx-ring.c

Posted by Willem at 1:39 PM 4 comments:

Labels: packet socket, PACKET_RX_RING

Wednesday, July 13, 2011

HOWTO: bind to a non local address (transparent proxy)

In certain situations, you may want to send packets as if they're coming from a different computer. Linux prevents such IP address spoofing by default, because the most well known use is as a malicious spoofing attack. Still, there are legitimate reasons. For instance, a transparent proxy intercepts traffic and replies in name of the original destination. Especially with larger sites, it is common to setup a virtual destination address and have a set of servers handle the load by mimicking this virtual host.

In Linux 2.6+, to spoof packets in IPv4, bind an INET socket to a non-local address, as in this straightforward example:

```
#include <errno.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define SPOOFPORT  80                // really, whichever you want
#define SPOOFADDR  ((214 << 24) + 1) // address to impersonate

int main(int argc, char **argv)
{
  struct sockaddr_in spoofaddr;
  int fd;

  fd = socket(PF_INET, SOCK_DGRAM, 0);
  if (fd == -1) {
    perror("socket");
    return 1;
  }

  memset(&spoofaddr, 0, sizeof(spoofaddr));
  spoofaddr.sin_family      = AF_INET;
  spoofaddr.sin_port        = htons(SPOOFPORT);
  spoofaddr.sin_addr.s_addr = htonl(SPOOFADDR);
  if (bind(fd, (void*) &spoofaddr, sizeof(spoofaddr))) {
    perror("bind");
    return 1;
  }

  if (close(fd)) {
    perror("close");
    return 1;
  }
  printf("OK\n");
  return 0;
}
```

When executed on most platforms, this piece of code will print

```
bind: Cannot assign requested address
```

To enable transparent proxy support in bind, two prerequisites must be met: (1) the application must have the `CAP_NET_ADMIN` capability and (2) the host must allow transparent proxying. The easiest way to enable the first is to start with superuser privileges. Obviously, drop these privileges as soon as they are no longer needed.

The second setting is configured through a `procsfs`. Make sure that `/proc/sys/net/ipv4/ip_nonlocal_bind` is set to 1.

Now, the code should execute succesfully and you can send to any host while masquerading as coming from the Department of Defense. Yes, that's the owner of 214.x.x.x. May want to change that macro.

Posted by Willem at 2:34 PM No comments:

Labels: bsd sockets, linux, socket programming, transparent proxy

Thursday, January 27, 2011

HOWTO: Enable console autologin on Ubuntu

This entry explains how you can automatically login to your Ubuntu machine.
*Warning: the following is a **security hazard** on network attached and publicly accessible machines*.
On throwaway development boxes, on the other hand, you sometimes need to reboot often and want to skip the annoying login. Only use this on inherently private machines: those that are physically secure and disconnected from the internet.

Traditional Linux

On traditionally configured linux machines, download mingetty from sourceforge, compile and install it and then edit `/etc/inittab`. For each console, replace lines such as

```
1:1:respawn:/etc/getty 9600 tty
```

with

```
1:1:respawn:/sbin/mingetty --autologin USERNAME tty1
```

To be automatically presented with a logged-in console on boot, also disable your graphical login managers. These are usually active on console 7.

Ubuntu

Ubuntu sometimes deviates from standard practice; the boot process is one example. First, to install mingetty, just run

```
sudo apt-get install mingetty
```

Ubuntu 9.04 inittab

Then, update the replacement of inittab. On Ubuntu 9.04, this file is replaced by a series of files in `/etc/event.d`. To automatically login on console `tty1`, edit `/etc/event.d/tty1`. Replace the use of `getty` in the last line from

```
exec /sbin/getty 38400 tty1
```

to

```
exec /sbin/mingetty --autologin USERNAME tty1
```

To automatically login on other consoles, be sure to replace tty1 with the correct name of the console.

Ubuntu 10.04 inittab

Ubuntu also changes its init process occasionally. In 10.04, the file `/etc/event.d/tty1` is replaced by `/etc/init/tty1.conf`. The changes are similar to those explained above.

Posted by Willem at 5:01 AM No comments:

Saturday, August 28, 2010

A Simple Python NodeVisitor Example

The Python `ast` module helps parsing python code into its abstract syntax tree and manipulating the tree. This is immensely useful, e.g., when you want to inspect code for safety or correctness. This page walks you through a very simple example to get you up and running quickly.

A Very Simple Parser

The first step to analyzing a program is parsing the textual source code into an in-memory walkable tree. Module `ast` takes away the hard task of implementing a generic lexer and python parser with the function `ast.parse()`. The module also takes care of walking the tree. All you have to do is implement analysis code for the type of statements you are want to analyze. To do this, you implement a class that derives from class `ast.NodeVisitor` and only override the member functions that are pertinent to your analysis.

NodeVisitor exposes callback member functions that the tree walker ("node visitor") calls when it encounters particular python statements, one for each type of statement in the language. For instance, `NodeVisitor.`**`visit_Import`** is called each time the waloker encounters an import statement.

Enough with the theory..

As code is often the best documentation, let's look at a parser that does one thing: print a message for each import statement

```
import ast

class FirstParser(ast.NodeVisitor):

    def __init__(self):
        pass

    def visit_Import(self, stmt_import):
      # retrieve the name from the returned object
      # normally, there is just a single alias
      for alias in stmt_import.names:
        print 'import name "%s"' % alias.name
        print 'import object %s % alias

      # allow parser to continue to parse the statement's children
      super(FirstParser, self).generic_visit(self, stmt_import)
```

For the code snippet "`import foo`", this produces

```
import name "foo"
import object <_ast.alias object at 0x7f05b871a690>
```

Implementing `visit_..` Callbacks

You can define callbacks of the form `visit_<type>` for each of the left-hand symbols and right-hand constructors defined in the abstract grammar for python. Thus, `visit_stmt` and `visit_Import` are both valid callbacks. Left-hand symbols may be abstract classes that are never called directly, however. Instead, their concrete implementations listed on the right are: `visit_stmt` is never called, but `visit_Import` implements a concrete type of statement and will be called for all import statements.

In the common case, when a node has no associated `visit_<type>` member, the parser calls the member `generic_visit`, which ensures that the walk recurses to the children of that node -- for which you may have a member defined, even if you did not define one for the node. When you override a member, that function is called and `generic_visit` is no longer called automatically. You are responsible for ensuring that the children are called, by calling `generic_visit` explicitly (unless you expressly intended to stop recursion) in your member.

Using the returned objects

Each callback function `visit_<type>(self, object)` returns with an object of a class particular to the given type. All classes derive from the abstract class ast.AST. As a result, each has a member `_fields`, along with members specific to the class. The `names` member shown in the first example is specific to `visit_Import`, for instance. Note that this corresponds to the argument of the `Import` constructor in the syntax. In general, I believe that these arguments are the class-specific members, although I could not find any definite documentation on this.

The `_fields` Member

The following snippet gives an example of how iterating of `_fields` returns all children of a node. Given the input "`a = b + 1`", the member function

```
def visit_BinOp(self, stmt_binop):
  for child in ast.iter_fields(stmt_expr):
    print 'child %s' % str(child)
```

generates the output

```
child ('left', <_ast.Name object at 0x7f05b871a710>)
child ('op', <_ast.Add object at 0x7f05b8715610>)
child ('right', <_ast.Num object at 0x7f05b871a750>)
```

For each child, the generator returns a tuple consisting of name and child object. A quick look at the abstract syntax grammar shows that indeed all child classes again correspond to symbols in the grammar: Name, Add and Num.

Calling the Parser

This brings us to the last step: how to actually pass input to the parser and generate output. Assuming you have a string containing Python code, this string is parsed into an in-memory tree and the tree walked with your callbacks using:

```
code = "a = b + 5"
tree = ast.parse(code)
parser = FirstParser()
parser.visit(tree)
```

A Warning on Modifying Code

Tree walking is not just useful for inspecting code, you can also use it to modify the parse tree. The reference documentation (see below) is very clear on the fact that you cannot use `NodeVisitor` for this purpose. Instead, derive from the

NodeTransformer class, whose members are expected to return a replacement object for each object with which they are called.

Further Reading

- The authoritative information sources is the Python ast module reference
- StackOverflow has a discussion with informative examples

Feedback

I wrote this mini tutorial, because I failed to find one when I first started using the ast module. That said, I'm no expert at it and not even a full-time Python programmer. If you spot errors or see room for improvement, don't hesitate to post a message.

Complete Example

The snippets above combine into the following example, which contains minor tweaks to avoid code duplication and improve readability:

```
import ast

class FirstParser(ast.NodeVisitor):

    def __init__(self):
        pass

    def continue(self, stmt):
        '''Helper: parse a node's children'''
        super(FirstParser, self).generic_visit(stmt)

    def parse(self, code):
        '''Parse text into a tree and walk the result'''
        tree = ast.parse(code)
        self.visit(tree)

    def visit_Import(self, stmt_import):
        # retrieve the name from the returned object
        # normally, there is just a single alias
        for alias in stmt_import.names:
            print 'import name "%s"' % alias.name
            print 'import object %s' % alias

        self.continue(stmt_binop)

    def visit_BinOp(self, stmt_binop):
        print 'expression: '
        for child in ast.iter_fields(stmt_binop):
            print '  child %s ' % str(child)

        self.continue(stmt_binop)


parser = FirstParser()
parser.parse('import foo')
parser.parse('a = b + 5')
```

Posted by Willem at 4:17 PM No comments:

Labels: ast, nodevisitor, parsing, python, syntax tree

Sunday, March 21, 2010

Pierce Firewall from within using netcat (e.g., for Bittorrent)

Opening ports in a firewall

If you find yourself behind a firewall that you cannot control, you often have no open network ports for others to contact you on. End-users generally only need this for peer to peer applications, such as Bittorrent and Skype.

Pretend to initiate an outbound connection using Netcat

Each time you make an outbound connection, the firewall creates a temporary opening to allow the other side to respond (say, Google to return your search results). You can exploit this feature to run Bittorrent or other servers. Pierce the firewall with a packet that originates from your computer and from the port that you want others to later contact you on (say, 6881 for Bittorrent). The easiest is to send a packet using netcat Using openbsd netcat, this worked for me:

```
nc -p 6881 www.google.com 80
```

Don't wait for a reply, just send the request, close netcat and open your real application. Note that the port will only remain open for a limited time if there is no traffic, so another computer has to make contact with yours practically immediately.

Limitations

It should work on most of the low-end routers that ISPs give you: these simply open up a port if you use it. More secure routers will only allow data between two specific computers: yours and the host you contacted (in the example, google). Then, you cannot use the opened port to serve peer to peer traffic. In short, YMMV.

The example uses openbsd netcat. Flags differ between implementations.

NB

I was quite surprised that I couldn't find a reference to this little trick online. I tried it out and it worked for me, but let me know if you see anything wrong with it.

Posted by Willem at 2:35 PM No comments:

Labels: firewall, nat, netcat, pierce

Home Older Posts

Subscribe to: Posts (Atom)