

ETH zürich spezialinf.ethz.ch @spezialinf

**ADRIAN PERRIG & TORSTEN HOEFLER**  
**Networks and Operating Systems (252-0062-00)**  
**Chapter 10: I/O Subsystems (2)**

### BREAKING FULL-DISK ENCRYPTION USING FIREWIRE

These days most owners of macOS laptops are using Apple's "TSM" to enable disk encryption on their machines. The main goal of this technique is to ensure that even if the laptop is stolen, the data is not accessible. However, as we will see in this slide, this is not always the case. In fact, it is possible to break the encryption and access the data on the disk.

As a previously mentioned, this is a complex task. It requires a lot of knowledge about the hardware and the operating system. However, it is possible to do it. The first step is to identify the hardware. In this case, we are looking for a FireWire controller. There are several ways to do this. One way is to use a hardware debugger. Another way is to use a software debugger. In this case, we are using a hardware debugger. The debugger allows us to access the hardware and see what is going on. We can see the data being written to the disk and see the encryption key. This is a very dangerous situation. If we have the key, we can decrypt the data. This is why it is important to be careful with I/O devices!

**BE CAREFUL WITH I/O DEVICES!**

Device	Model	Manufacturer	Serial Number
0x00000000	0x00000000	0x00000000	0x00000000
0x00000001	0x00000001	0x00000001	0x00000001
0x00000002	0x00000002	0x00000002	0x00000002
0x00000003	0x00000003	0x00000003	0x00000003
0x00000004	0x00000004	0x00000004	0x00000004
0x00000005	0x00000005	0x00000005	0x00000005
0x00000006	0x00000006	0x00000006	0x00000006
0x00000007	0x00000007	0x00000007	0x00000007
0x00000008	0x00000008	0x00000008	0x00000008
0x00000009	0x00000009	0x00000009	0x00000009
0x0000000A	0x0000000A	0x0000000A	0x0000000A
0x0000000B	0x0000000B	0x0000000B	0x0000000B
0x0000000C	0x0000000C	0x0000000C	0x0000000C
0x0000000D	0x0000000D	0x0000000D	0x0000000D
0x0000000E	0x0000000E	0x0000000E	0x0000000E
0x0000000F	0x0000000F	0x0000000F	0x0000000F
0x00000010	0x00000010	0x00000010	0x00000010
0x00000011	0x00000011	0x00000011	0x00000011
0x00000012	0x00000012	0x00000012	0x00000012
0x00000013	0x00000013	0x00000013	0x00000013
0x00000014	0x00000014	0x00000014	0x00000014
0x00000015	0x00000015	0x00000015	0x00000015
0x00000016	0x00000016	0x00000016	0x00000016
0x00000017	0x00000017	0x00000017	0x00000017
0x00000018	0x00000018	0x00000018	0x00000018
0x00000019	0x00000019	0x00000019	0x00000019
0x0000001A	0x0000001A	0x0000001A	0x0000001A
0x0000001B	0x0000001B	0x0000001B	0x0000001B
0x0000001C	0x0000001C	0x0000001C	0x0000001C
0x0000001D	0x0000001D	0x0000001D	0x0000001D
0x0000001E	0x0000001E	0x0000001E	0x0000001E
0x0000001F	0x0000001F	0x0000001F	0x0000001F

ETH zürich spezialinf.ethz.ch @spezialinf

### Our Small Quiz

- True or false (raise hand)
  - Open files are part of the process' address-space
  - Unified buffer caches improve the access times
  - A partition table can unify the view of multiple disks
  - Unix enables to bind arbitrary file systems to arbitrary locations
  - The virtual file system interface improves modularity of OS code
  - Programmed I/O is efficient for the CPUs
  - DMA enables devices to access virtual memory of processes
  - IOMMUs enable memory protection for devices
  - IOMMUs improve memory access performance
  - First level interrupt handlers process the whole request from the hardware
  - Software interrupts reduce the request processing latency
  - Deferred procedure calls execute second-level interrupt handlers

ETH zürich spezialinf.ethz.ch @spezialinf

### The I/O subsystem

ETH zürich spezialinf.ethz.ch @spezialinf

### Generic I/O functionality

- Device drivers essentially move data to and from I/O devices
  - Abstract hardware
  - Manage asynchrony
- OS I/O subsystem includes generic functions for dealing with this data
  - Such as...

ETH zürich spezialinf.ethz.ch @spezialinf


### The I/O Subsystem

- Caching** - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- Spooling** - hold output for a device
  - If device can serve only one request at a time
  - E.g., printing

ETH zürich spezialinf.ethz.ch @spezialinf


### The I/O Subsystem

- Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- Buffering** - store data in memory while transferring between devices or memory
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain "copy semantics"

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth


## Naming and Discovery

- What are the devices the OS needs to manage?
  - Discovery (bus enumeration)
  - Hotplug / unplug events
  - Resource allocation (e.g., PCI BAR programming)
- How to match driver code to devices?
  - Driver instance  $\neq$  driver module
  - One driver typically manages many models of device
- How to name devices inside the kernel?
- How to name devices outside the kernel?

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Matching drivers to devices


- Devices have unique (model) identifiers
  - E.g., PCI vendor/device identifiers
- Drivers recognize particular identifiers
  - Typically a list...
- Kernel offers a device to each driver in turn
  - Driver can "claim" a device it can handle
  - Creates driver instance for it.

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Naming devices in the Unix kernel


(Actually, naming *device driver instances*)

- Kernel creates identifiers for
  - Block devices
  - Character devices
    - [ *Network devices – see later...* ]
- Major device number:
  - Class of device (e.g., disk, CD-ROM, keyboard)
- Minor device number:
  - Specific device within a class

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth


## Unix Block Devices

- Used for "structured I/O"
  - Deal in large "blocks" of data at a time
- Often look like files (seekable, mappable)
  - Often use Unix' shared buffer cache
- Mountable:
  - File systems implemented above block devices

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Character Devices

- Used for "unstructured I/O"
  - Byte-stream interface – no block boundaries
  - Single character or short strings get/put
  - Buffering implemented by libraries
- Examples:
  - Keyboards, serial lines, mice
- Distinction with block devices somewhat arbitrary...

ETH zürich  [spoc.inf.ethz.ch](http://spoc.inf.ethz.ch)  
@spoc\_eth

## Naming devices outside the kernel

- Device files: special type of *file*
  - Inode encodes <type, major num, minor num>
  - Created with `mknod()` system call
- Devices are traditionally put in `/dev`
  - `/dev/sda` – First SCSI/SATA/SAS disk
  - `/dev/sda5` – Fifth partition on the above
  - `/dev/cdrom0` – First DVD-ROM drive
  - `/dev/ttyS1` – Second UART

**Pseudo-devices in Unix**

- Devices with no hardware!
- Still have major/minor device numbers. Examples:

```

/dev/stdin
/dev/kmem
/dev/random
/dev/null
/dev/loop0
    
```

etc.

**Old-style Unix device configuration**

- All drivers compiled into the kernel
- Each driver probes for any supported devices
- System administrator populates /dev
  - Manually types `mknod` when a new device is purchased!
- Pseudo devices similarly hard-wired in kernel

**Linux device configuration today**

- Physical hardware configuration readable from /sys
  - Special fake file system: `sysfs`
  - Plug events delivered by a special socket
- Drivers dynamically loaded as kernel modules
  - Initial list given at boot time
  - User-space daemon can load more if required
- /dev populated dynamically by `udev`
  - User-space daemon which polls /sys

**Interface to network I/O**

**Unix interface to network I/O**

- You already know the data path
  - BSD sockets
  - `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`, etc.
- Have not yet seen:
  - Device driver interface
  - Configuration
  - Routing

**Software routing**

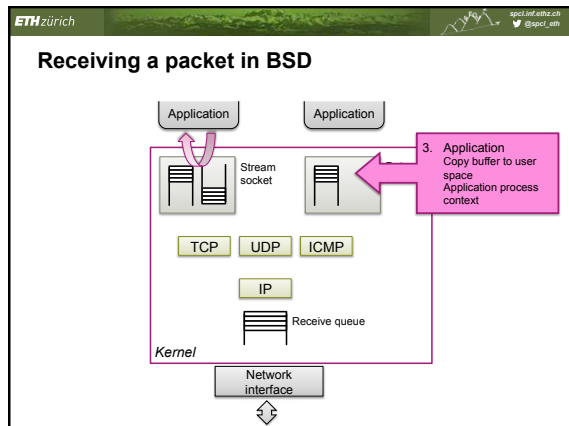
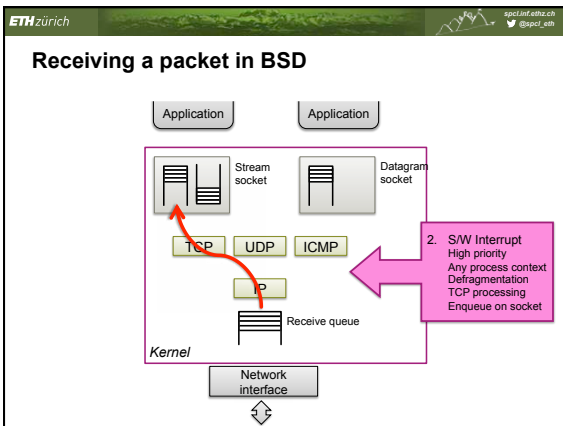
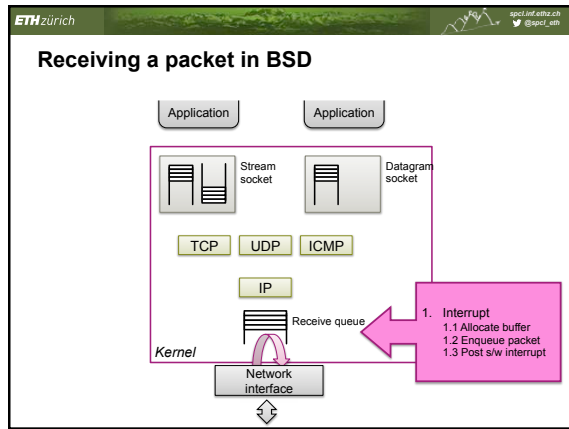
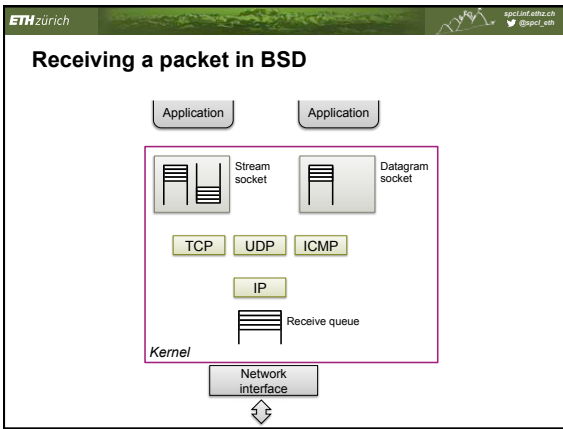
```

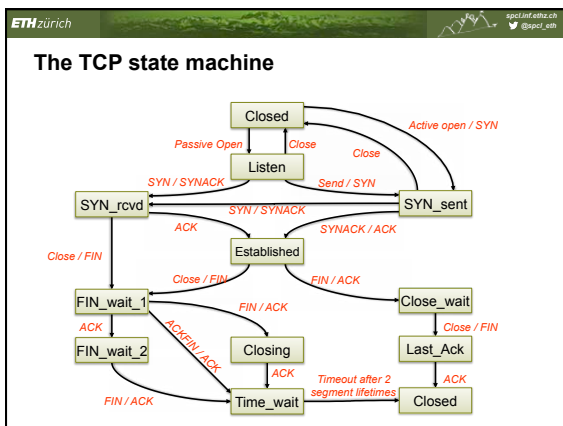
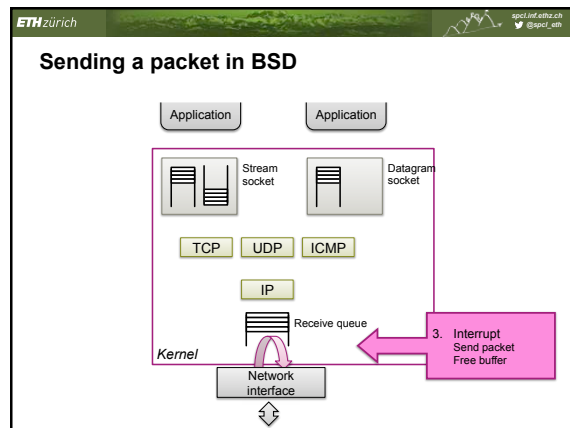
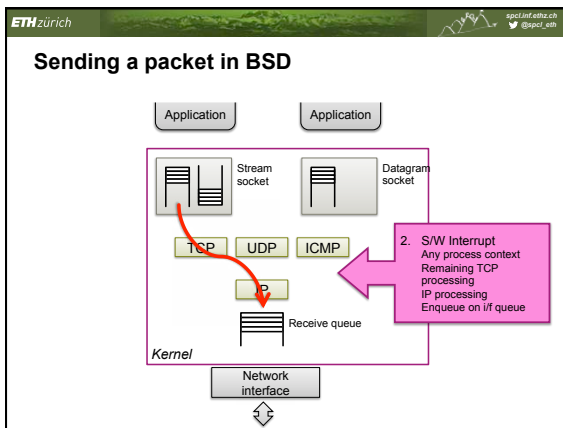
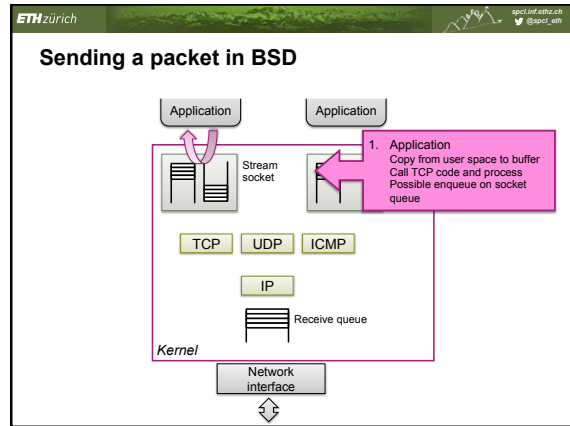
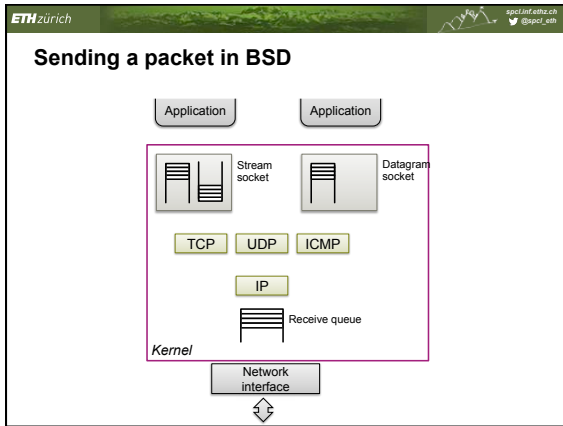
graph TD
    subgraph User_Space [User space]
        RD[Routing daemon]
    end
    subgraph Kernel_Space [Kernel space]
        PS[Protocol stack]
        FIB[FIB (forwarding information base)]
    end
    RD <--> PS
    RD <--> FIB
    PS <--> FIB
    PS <--> Network[Network]
    FIB <--> Network
    
```

- OS protocol stacks include routing functionality
- Routing protocols typically in a user-space daemon
  - Non-critical
  - Easier to change
- Forwarding information typically in kernel
  - Needs to be fast
  - Integrated into protocol stack

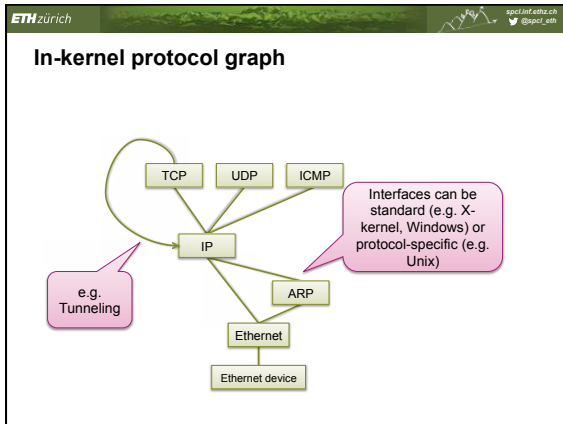
# Network stack implementation

- ## Networking stack
- Probably most important peripheral
    - GPU is increasingly not a peripheral
    - Disk interfaces look increasingly like a network
  - But...
    - NO standard OS textbook talks about the network stack!
  - Good references:
    - The 4.BSD book (for Unix at least)
    - George Varghese: "Network Algorithms" (up to a point)





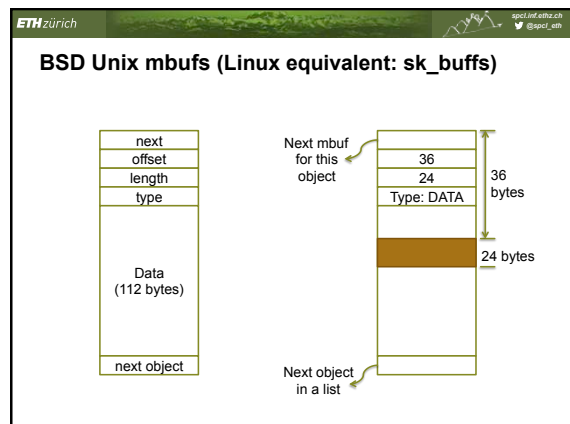
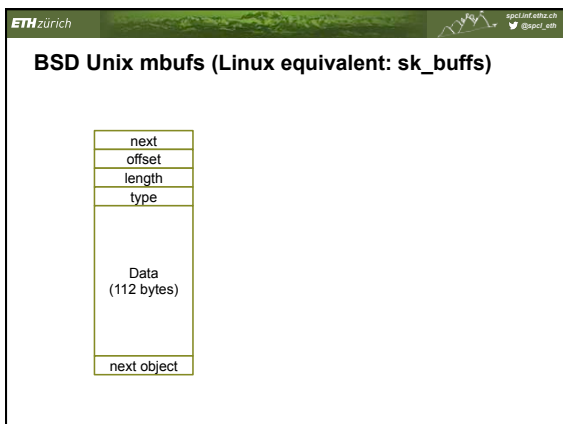
- ### OS TCP state machine
- **More complex! Also needs to handle:**
    - Congestion control state (window, slow start, etc.)
    - Flow control window
    - Retransmission timeouts
    - Etc.
  - **State transitions triggered when:**
    - User request: `send`, `recv`, `connect`, `close`
    - Packet arrives
    - Timer expires
  - **Actions include:**
    - Set or cancel a timer
    - Enqueue a packet on the transmit queue
    - Enqueue a packet on the socket receive queue
    - Create or destroy a *TCP control block*



- ### Protocol graphs
- Graph nodes can be:
- **Per-protocol (handle all flows)**
    - Packets are "tagged" with demux tags
  - **Per-connection (instantiated dynamically)**
    - Multiple *interfaces* as well as connections
    - Ethernet ↔ Ethernet ⇒ bridging
    - IP ↔ IP ⇒ IP routing

### Memory management

- ### Memory management
- **Problem: how to ship packet data around**
  - **Need a data structure that can:**
    - Easily add, remove headers
    - Avoid copying lots of payload
    - Uniformly refer to half-defined packets
    - Fragment large datasets into smaller units
  - **Solution:**
    - Data is held in a linked list of "buffer structures"



**ETH zürich** spezial@ethz.ch @spoc\_eth

### Case Study: Linux 3.x

- **Implementing a simple protocol over Ethernet**
- **Why?**
  - You want to play with networking equipment (well, RAW sockets are easier)
  - You want to develop specialized protocols  
E.g., application-specific "TCP"  
E.g., for low-latency cluster computing
  - You'll understand how it works!

**ETH zürich** spezial@ethz.ch @spoc\_eth

### Sending Data in Linux 3.x

- Many layers
- Most use the sk\_buff struct

**ETH zürich** spezial@ethz.ch @spoc\_eth

### Register a receive hook

- **Fill packet\_type struct:**
  - .type = your ethertype
  - .func = your receive function
- **Receive handler recv\_hook(...)**
  - Gets sk\_buff, packet\_type, net\_device, ...
  - Called for each incoming frame
  - Reads skb->data field and processes protocols

Receive hook table:

0x0800	IPv4 hdr.
0x8864	PPPOE hdr.
0x8915	RoCE hdr.
...	

**ETH zürich** spezial@ethz.ch @spoc\_eth

### Interaction with applications

- **Socket Interface**
  - Need to implement handlers for connect(), bind(), listen(), etc.
- **Call sock\_register(struct net\_proto\_family\*)**
  - Register a protocol family
  - Enables user to create socket of this type

**ETH zürich** spezial@ethz.ch @spoc\_eth

### Anatomy of struct sk\_buff

- **Called "skb" in Linux jargon**
  - Allocate via alloc\_skb() (or dev\_alloc\_skb() if in driver)
  - Free with kfree\_skb() (dev\_kfree\_skb())
  - Use pskb\_may\_pull(skb, len) to check if data is available
  - skb\_pull(skb, len) to advance the data pointer  
... it even has a webpage! <http://www.skbuff.net/>

**ETH zürich** spezial@ethz.ch @spoc\_eth

### SKB Fields

- **Double-linked list, each skb has .next/.prev**
  - .data contains payload (size of data field is set by skb\_alloc)
  - .sk is the socket this skb is owned by
  - .mac\_header, .network\_header, .transport\_header contain headers of various layers
  - .dev is the device this skb uses
  - ... 58 member fields total

**ETH zürich** spci.inf.ethz.ch @spci\_eth

### Case Study: TCP Fragmenting

- Linux <2.0.32:
  - Two fragments:
    - #1
      - Offset: 0
      - Length: 100
    - #2
      - Offset 100
      - Length: 100

```

// Determine the position of this fragment.
end = offset + iph->tot_len - ihl; #1: 100, #2: 200
// Check for overlap with preceding fragment, and, if needed,
// align things so that any overlaps are eliminated.
if (prev != NULL && offset < prev->end) {
    i = prev->end - offset;
    offset += i; /* ptr into datagram */
    ptr += i; /* ptr into fragment data */
}
// initialize segment structure
fp->offset = offset; #1: 0, #2: 100
fp->end = end; #1: 100, #2: 200
fp->len = end - offset; #1: 100, #2: 100
.... // collect multiple such fragments in queue
// process each fragment
if (count+fp->len > skb->len) {
    error_to_big;
}
memcpy((ptr + fp->offset), fp->ptr, fp->len);
count += fp->len;
fp = fp->next;
    
```

**ETH zürich** spci.inf.ethz.ch @spci\_eth

### Case Study: TCP Fragmenting

- Linux <2.0.32:
  - Two fragments:
    - #1
      - Offset: 0
      - Length: 100
    - #2
      - Offset 10
      - Length: 20

```

// Determine the position of this fragment.
end = offset + iph->tot_len - ihl; #1: 100, #2: 30
// Check for overlap with preceding fragment, and, if needed,
// align things so that any overlaps are eliminated.
if (prev != NULL && offset < prev->end) {
    i = prev->end - offset;
    offset += i; /* ptr into datagram */
    ptr += i; /* ptr into fragment data */
}
// initialize segment structure
fp->offset = offset; #1: 0, #2: 100
fp->end = end; #1: 100, #2: 30
fp->len = end - offset; #1: 100, #2: -70
.... // collect multiple such fragments in queue
// process each fragment
if (count+fp->len > skb->len) {
    error_to_big;
}
memcpy((ptr + fp->offset), fp->ptr, fp->len);
count += fp->len;
fp = fp->next;
    
```

*(size\_t)-70 = 4294967224*

**ETH zürich** spci.inf.ethz.ch @spci\_eth

### Case Study: TCP Fragmenting

**Windows**

A fatal exception 0E has occurred at 0028:C1891963 in UXD ctpci9x(05) + 00001053. The current application will be terminated.

- Press any key to terminate the current application.
- Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue \_

**ETH zürich** spci.inf.ethz.ch @spci\_eth

### 2.0.32 ... that's so last century!

Security Technical Center - Security Bulletin - Microsoft Security Bulletin MS09-050

#### Microsoft Security Bulletin MS09-050 - Critical

##### Vulnerabilities in SMBv2 Could Allow Remote Code Execution (975517)

Published: Tuesday, October 13, 2009 | Updated: Wednesday, October 14, 2009

Version: 1.1

**Executive Summary**

This security update resolves one publicly disclosed and two privately reported vulnerabilities in Server Message Block Version 2 (SMB2). The most severe of the vulnerabilities could allow remote code execution. **Update availability: critical SMBv2 updates.** Computer running the Server service. Fewest best practices and standard default firewall configurations can help protect resources from attacks that originate from outside the enterprise perimeter. Best practices recommend that systems that are connected to the Internet have a minimal number of ports exposed.

This security update is rated Critical for supported editions of Windows Vista and Windows Server 2008. For more information, see the subsection, **Affected and Non-Affected Software**, in this section.

The security update addresses the vulnerabilities by correctly validating the fields inside the SMB2 packets, correcting the way that SMB handles the command value in SMB packets, and correcting the way that SMB parses specially crafted SMB packets. For more information about the vulnerabilities, see the Frequently Asked Questions (FAQ) subsection for the specific vulnerability entry under the next section, **Vulnerability Information**.

This security update also addresses the vulnerability first described in Microsoft Security Advisory 975487.

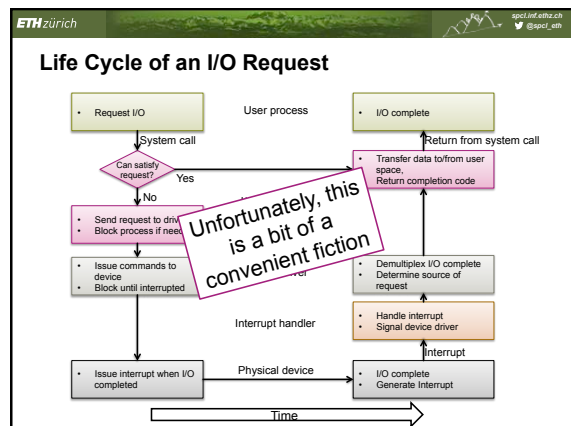
**Recommendation.** The majority of customers have automatic updating enabled and will not need to take any action because this security update will be downloaded and installed automatically. Customers who have not enabled automatic updating need to check for updates and install this update manually. For information about specific configuration options in automatic updating, see Microsoft Knowledge Base article 554622.

For administrators and enterprise installations, or end users who want to install this security update manually, Microsoft recommends that customers apply the update immediately using update management software, or by checking for updates using the Microsoft Update service.

See also the section, **Detection and Deployment Tools and Guidance**, later in this bulletin.

**ETH zürich** spci.inf.ethz.ch @spci\_eth

### Performance issues





ETH zürich spezial@ethz.ch @spoc\_eth

### Consider 10 Gb/s Ethernet



ETH zürich spezial@ethz.ch @spoc\_eth

### At full line rate for 1 x 10Gb port

- **~1GB (gigabyte) per second**
  - ⇒ ~ 700k full-size Ethernet frames per second
  - ⇒ At 2GHz, must process a packet in ≤ 3000 cycles
- **This includes:**
  - IP and TCP checksums
  - TCP window calculations and flow control
  - Copying packet to user space

ETH zürich spezial@ethz.ch @spoc\_eth


### A few numbers...

- **L3 cache miss (64-byte lines) ≈ 300 cycles**
  - ⇒ At most 10 cache misses per packet
  - Note: DMA ensures cache is cold for the packet!
- **Interrupt latency ≈ 500 cycles**
  - Kernel entry/exit
  - Hardware access
  - Context switch / DPC
  - Etc.

ETH zürich spezial@ethz.ch @spoc\_eth

### Plus...


- **You also have to send packets.**
  - Card is full duplex ⇒ can send at 10Gb/s
- **You have to do something useful with the packets!**
  - Can an application can make use of 1.5kB of data every 1000 machine cycles or so?
- **This card has two 10Gb/s ports.**



ETH zürich spezial@ethz.ch @spoc\_eth

### And Plus ...

- **And if you thought that was fast ...**
  - Mellanox 100 Gb/s Adapter
  - Impossible to use without advanced features
    - RDMA*
    - SR-IOV*
    - TOE*
    - Interrupt coalescing*



ETH zürich spezial@ethz.ch @spoc\_eth

### What to do?

- **TCP offload (TOE)**
  - Put TCP processing into hardware on the card
- **Buffering**
  - Transfer lots of packets in a single transaction
- **Interrupt coalescing / throttling**
  - Don't interrupt on every packet
  - Don't interrupt at all if load is very high
- **Receive-side scaling**
  - Parallelize: direct interrupts and data to different cores

**Linux New API (NAPI)**

- **Mitigate interrupt pressure**
  1. Each packet interrupts the CPU
  2. Vs. CPU polls driver
    - NAPI switches between the two!
- **NAPI-compliant drivers**
  - Offer a poll() function
  - Which calls back into the receive path ...

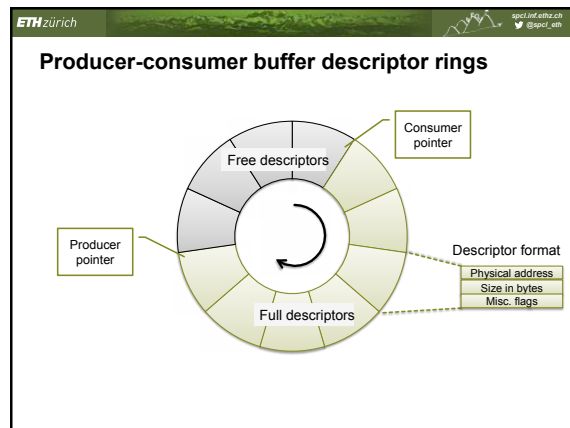
**Linux NAPI Balancing**

- **Driver enables polling with netif\_rx\_schedule(struct net\_device \*dev)**
  - Disables interrupts
- **Driver deactivates polling with netif\_rx\_complete(struct net\_device \*dev)**
  - Re-enable interrupts.
- → but where does the data go???

**Buffering**

**Key ideas:**

- **Decouple sending and receiving**
  - Neither side should wait for the other
  - Only use interrupts to unblock host
- **Batch together requests**
  - Spread cost of transfer over several packets

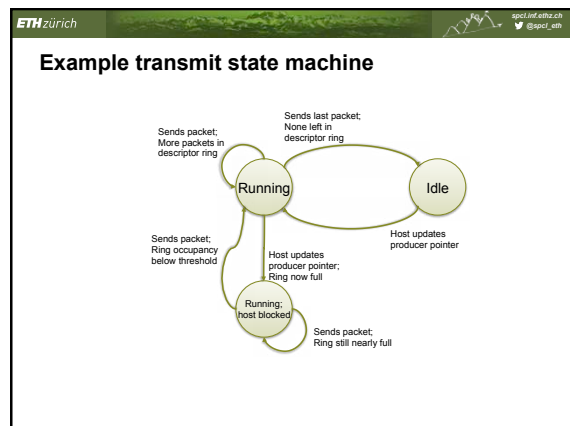


**Buffering for network cards**

**Producer, consumer pointers are NIC registers**

- **Transmit path:**
  - Host updates producer pointer, adds packets to ring
  - Device updates consumer pointer
- **Receive path:**
  - Host updates consumer pointer, adds empty buffers to ring
  - Device updates producer pointer, fills buffers with received packets.

**More complex protocols are possible...**



**Transmit interrupts**

- **Ring empty**
  - ⇒ all packets sent
  - ⇒ device going idle
- **Ring occupancy drops**
  - ⇒ host can now send again
  - ⇒ device continues running

```

    graph TD
      Idle((Idle)) -- "Host updates producer pointer" --> Running((Running))
      Running -- "Sends packet, More packets in descriptor ring" --> Running
      Running -- "Sends last packet, None left in descriptor ring" --> Idle
      Running -- "Host updates producer pointer, Ring now full" --> Blocked((Running, host blocked))
      Blocked -- "Sends packet, Ring still nearly full" --> Blocked
      Blocked -- "Sends packet, Ring occupancy below threshold" --> Running
  
```

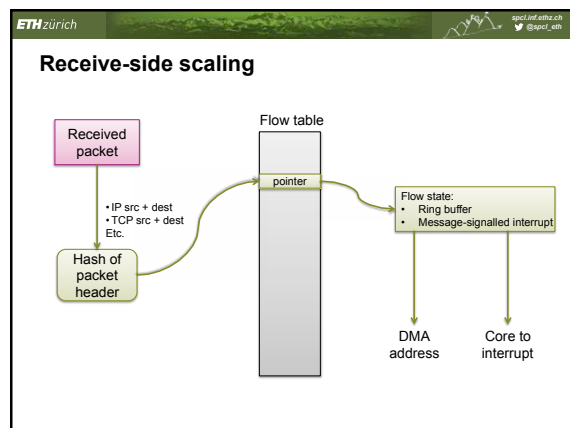
Exercise: devise a similar state machine for receive!

**Buffering summary**

- **DMA used twice**
  - Data transfer
  - Reading and writing descriptors
- **Similar schemes used for any fast DMA device**
  - SATA/SAS interfaces (such as AHCI)
  - USB2/USB3 controllers
  - etc.
- **Descriptors send ownership** of memory regions
- **Flexible – many variations possible:**
  - Host can send lots of regions in advance
  - Device might allocate out of regions, send back subsets
  - Buffers might be used out-of-order
- **Particularly powerful with multiple send and receive queues...**

**Receive-side scaling**

- **Insight:**
  - Too much traffic for one core to handle
  - Cores aren't getting any faster
  - ⇒ Must parallelize across cores
- **Key idea: handle different flows on different cores**
  - But: how to determine flow for each packet?
  - Can't do this on a core: same problem!
- **Solution: demultiplex on the NIC**
  - DMA packets to per-flow buffers / queues
  - Send interrupt only to core handling flow



**Receive-side scaling**

- **Can balance flows across cores**
  - Note: doesn't help with one big flow!
- **Assumes:**
  - $n$  cores processing  $m$  flows is faster than one core
- **Hence:**
  - Network stack and protocol graph must *scale* on a multiprocessor.
- **Multiprocessor scaling: topic for later**

**Tomorrow**

- **Virtual machines**
- **Multiprocessor operating systems**