Passive Measurement with Libtrace

Shane Alcock

Outline

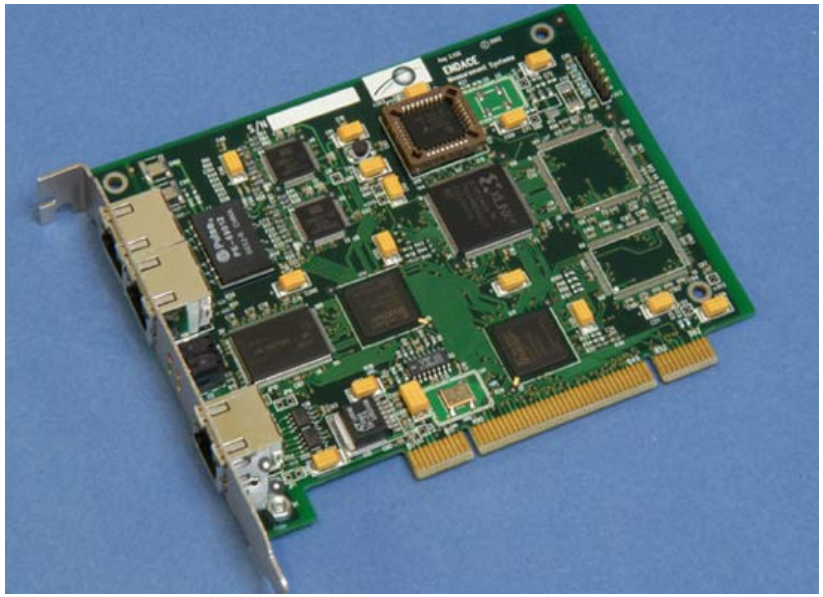
- Introduction to Passive Measurement
- Programming Issues in Passive Measurement
- Introducing Libtrace
- Acquiring and Installing Libtrace
- Libtrace Basics: URIs and BPF filters
- Libtrace Tools
- Libtrace API and Examples
- Assignment

Passive Measurement

- Use existing network traffic to analyse network behaviour
  - No artificial “measurement” traffic
- Can be divided into two principal steps
  - Capture – reading data off the network
  - Analysis – decoding and processing the data
- We're going to focus on measurement at the packet level

Packet Capture

- Hardware
  - Endace DAG cards
  - Intel DPDK
- Software
  - PCAP (tcpdump)
- Kernel
  - Linux native



Packet Capture

- A header is prepended to each captured packet
  - Timestamps
  - Packet length
- Header structure differs for each capture format
  - Timestamp format can be different too

Packet Capture

- Example: ERF header used by DAG

Timestamp		
Timestamp (floating point)		
Frame Type	Flags	Record Length
Loss Counter		Wire Length

- Example: PCAP header

Timestamp	
Timestamp (microseconds)	
Capture Length	Wire Length

Packet Traces

- Captured packets can be written to disk to create a trace
  - Packets are in chronological order
  - Capture format header is retained on each packet

- Analysis is repeatable
  - Errors in analysis technique can be corrected
  - Interesting behaviour can be investigated further

- Collaboration with other researchers
  - WITS - http://www.wand.net.nz/wits/
  - Datcat - http://www.datcat.org/
  - CRAWDAD - http://crawdad.cs.dartmouth.edu/

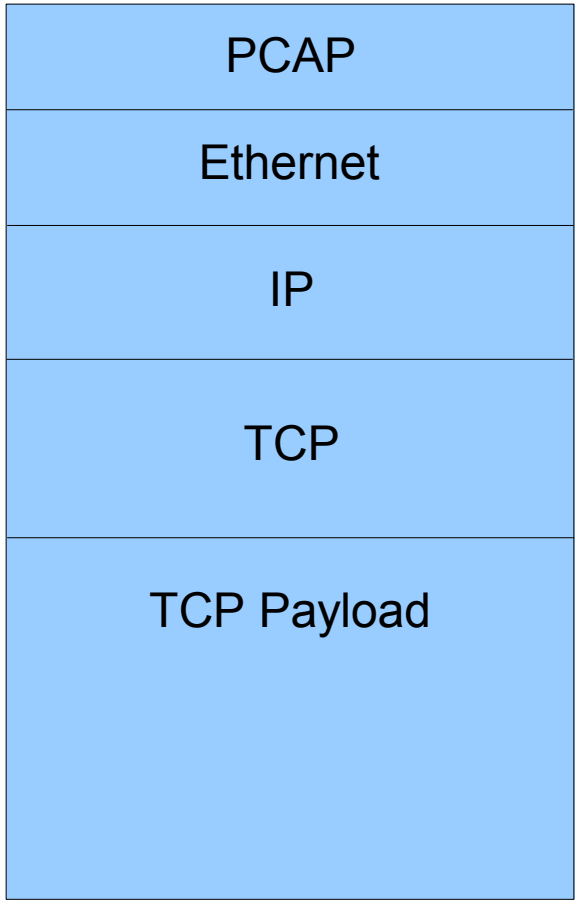
Capture Techniques

- Full payload capture
  - All of the packet payload is retained
  - Simple to implement
  - Investigating application behaviour is easier

- Disadvantages
  - Privacy concerns due to capturing user data
  - Trace files are extremely large

Capture Techniques

- Example – blue area represents the captured data

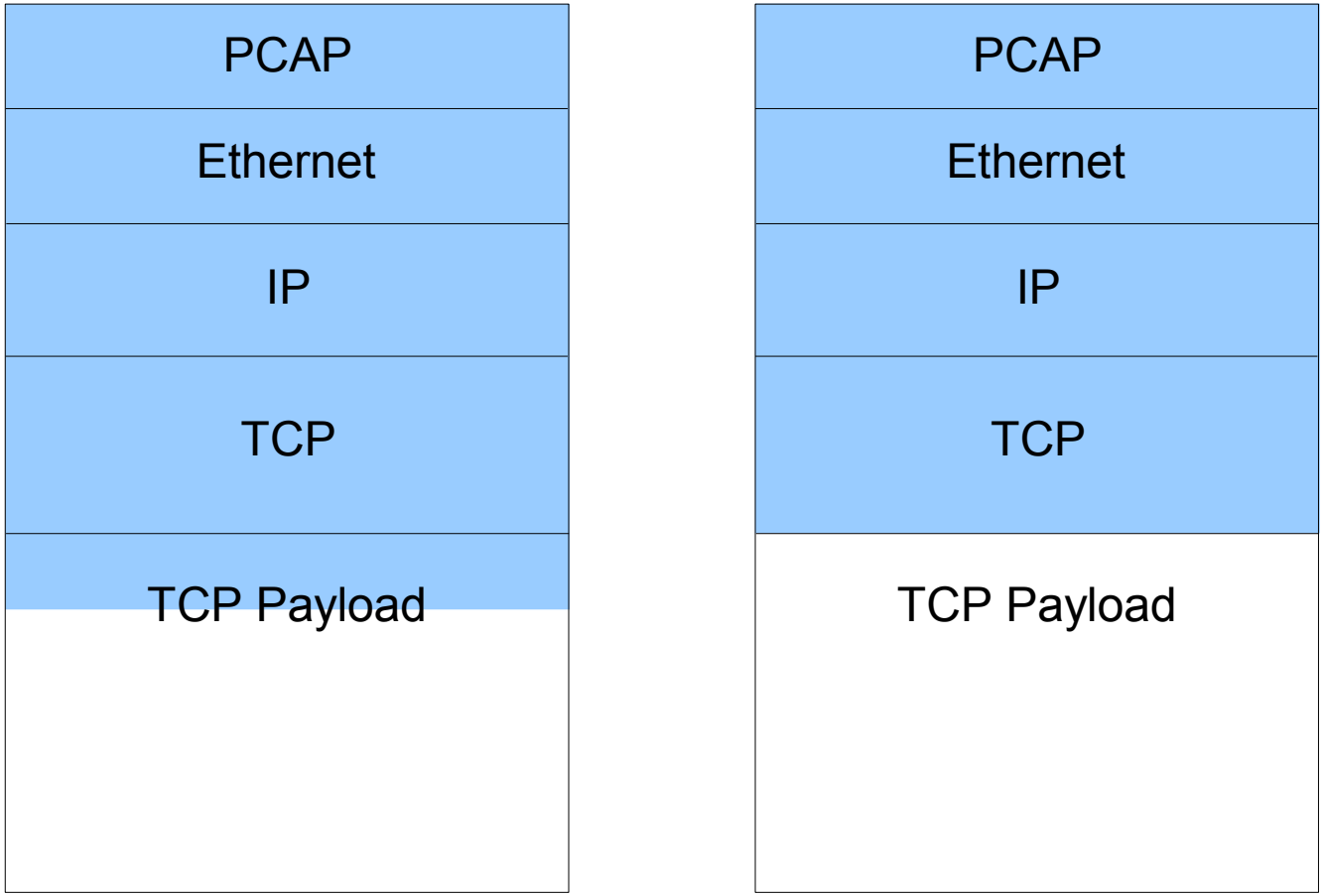


Capture Techniques

- Header capture
  - Captured packets are truncated (snapped) to remove user payload
    - Fixed length snapping vs header-based snapping
  - Traces require less space
  - Most pertinent information is in the headers

Packet Traces

- Example – fixed length (left) vs header snapping (right)



Passive Analysis

- Simple examples
  - Counting packets or bytes
  - Examining TCP/IP headers
- Advanced ideas
  - TCP object extraction
  - Traffic classification
  - Application-specific analysis, e.g. YouTube performance
  - Visualisation

Passive Analysis

- Real-time
  - Capture process reads straight off a network interface
  - Performance is critical
  - Most practical applications are real-time
    - e.g. anomaly detection, IDS, visualisation

- Off-line
  - Replace the capture step with reading from a trace file
  - Best for resource-intensive analysis
  - Many research applications can be done solely off-line

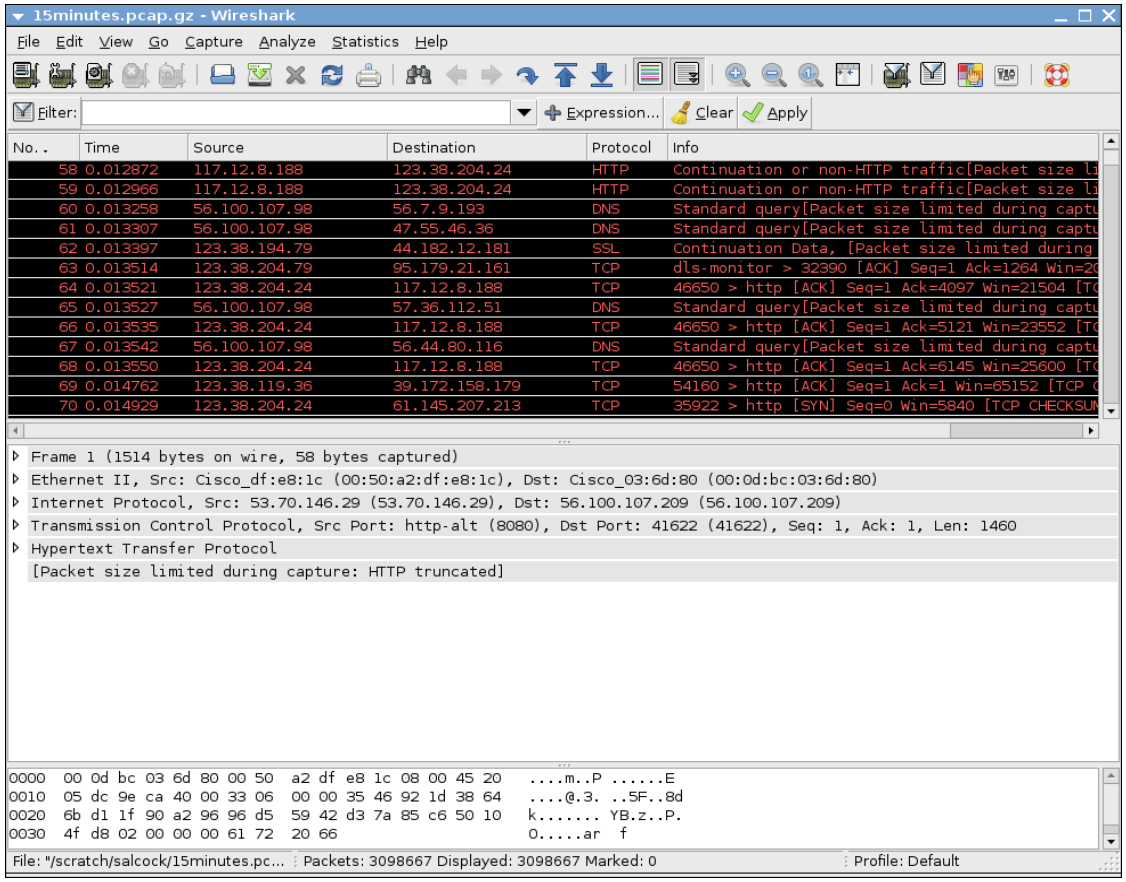
Analysis Software

- Use existing tools
  - Examples: wireshark, tcptrace
  - Designed to perform a specific set of tasks

- Develop new analysis tools
  - Particularly common in research

Analysis Software

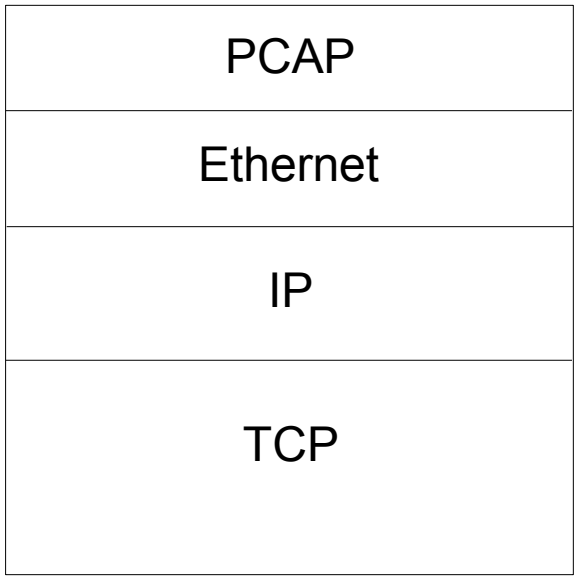
- Example of an existing tool: wireshark



Development Issues

- Aim is to count packets using TCP port 80
  - Should be easy, right?

- Standard TCP/IP packet captured using PCAP from an Ethernet link



Development Issues

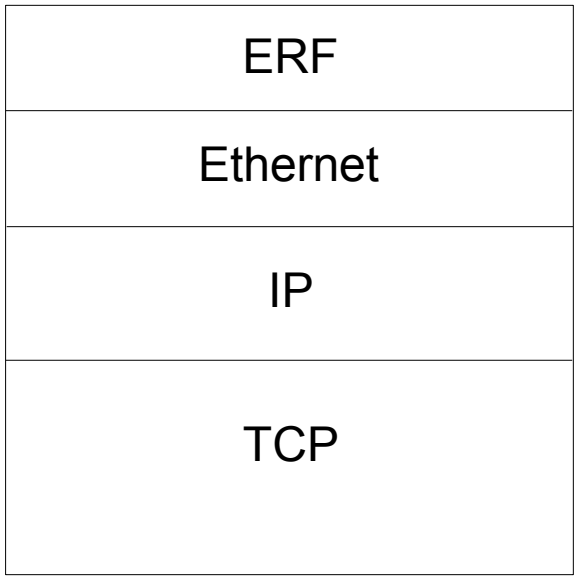
- The general case is simple
  - Step through the preceding headers to reach the TCP header
    - Be careful of the variable length IP header!
  - Check the port numbers inside the TCP header
  - Increment counter if necessary
  - Move onto next packet
    - PCAP header will tell us how far we need to skip ahead

But...

- What about the special cases?
  - The packet isn't a TCP packet, e.g. UDP or ICMP
  - The packet isn't an IP packet, e.g. ARP
  - The packet was truncated before the TCP header
  - The packet was truncated part-way through the TCP header
  - The packet was fragmented
    - TCP header could be in a different fragment
  - Note that this is not a comprehensive list!

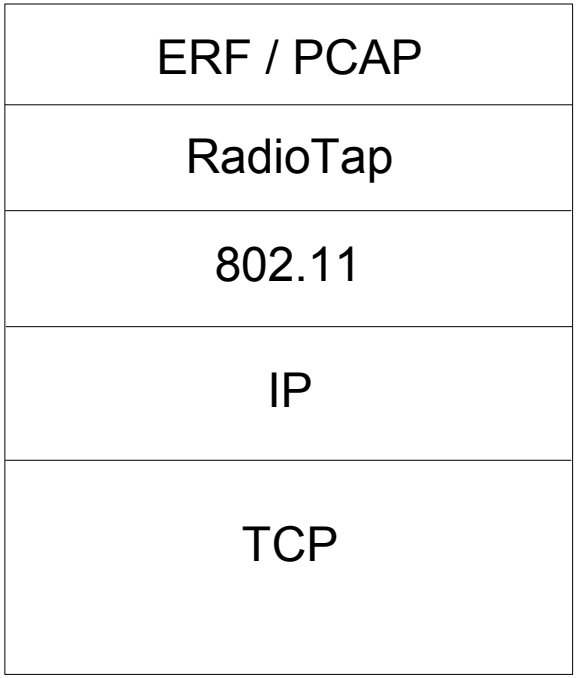
Format Changes

- Try our analysis on another trace set
  - What if the traces use the ERF format instead of PCAP?
  - Update program to support new capture format



Different Link Types

- Applying our analysis to a trace from a wireless link
  - Need to add code to detect and skip over 802.11 headers
  - Still need to keep our old code for Ethernet as well

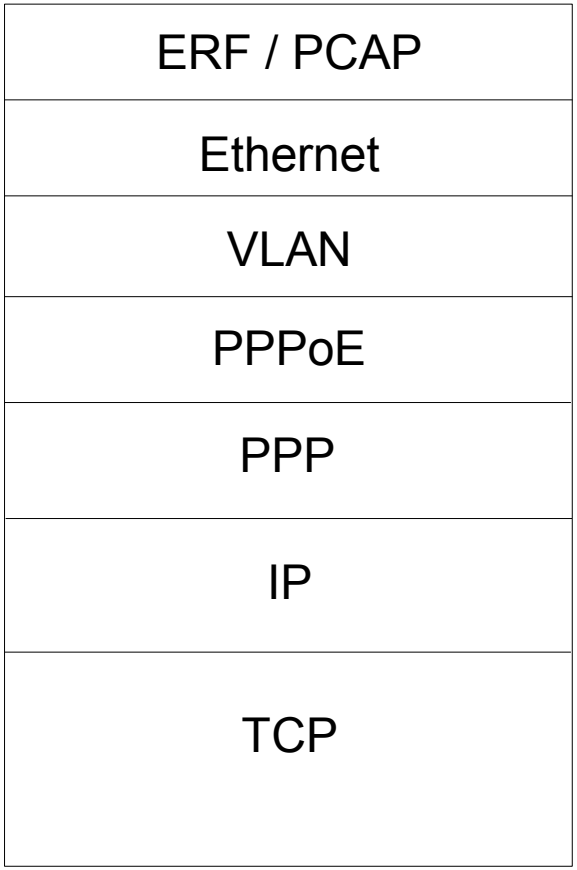


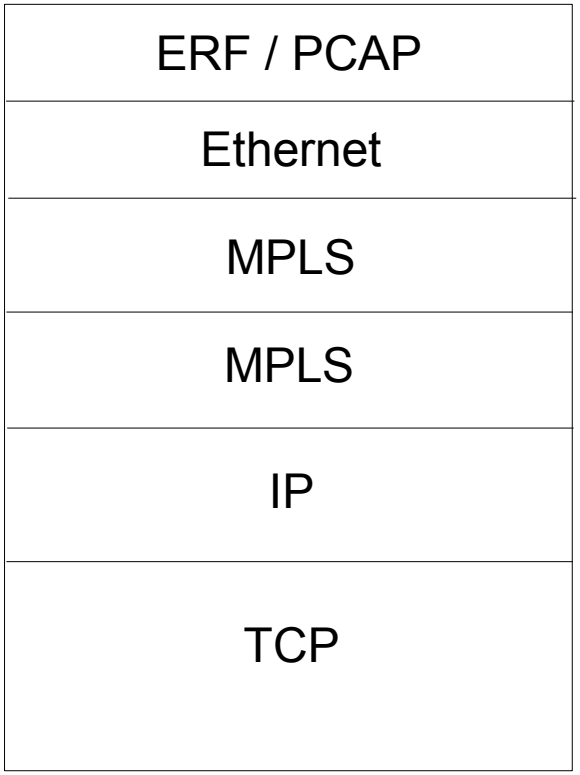
Wireless is Hard

- Wireless introduces an entirely new set of problems
  - 802.11 header varies in length
  - RadioTap header is not always present
    - Might be an entirely different header altogether, e.g. Prism
    - Might be no header at all before the 802.11 header
  - Frame corruption
  - Fragmentation can also occur at the 802.11 level
  - Once again, not a comprehensive list

More Link Types

- Other link layer protocols





Go Live!

- What about running our analysis on a live capture?
  - Live capture APIs add an extra level of complexity
  - Buffer management
  - Code needs to be efficient

Summary

- Developing a portable analysis tool is very difficult
  - Subtle differences between each format header
  - Link layer encapsulation is a nightmare
  - Live capture formats are particularly difficult to code
  - Huge variety of special cases and banana skins

- Wouldn't it be nice if someone...
  - did all the tricky programming for us
  - wrapped it in a nice API that abstracted away all the nasty details
  - gave it all away for free
  - was willing to show you how to use it

Say Hello to Libtrace

- Packet capture and analysis library
- Developed by WAND (University of Waikato)
  - Written in C, but there are also Python and Ruby bindings
- Design aimed to resolve all the issues discussed earlier
  - Make passive analysis simple and reduce code replication
- Supports reading and writing of trace files

Libtrace Features

- Capture format agnostic
  - The same libtrace program works on any supported capture format
  - No difference between live and off-line capture formats
  - Developmental advantages
    - Analysis programs can be tested off-line before running live
  - Input and output formats for the same program can be different
    - Libtrace will perform the conversion internally for you

Libtrace Features

- Protocol details are dealt with internally
  - Direct access to each protocol layer
    - e.g. **`trace_get_layer3()`** jumps straight to the IP header
  - Handles a variety of link layer protocols including ...
    - Ethernet
    - 802.11 wireless
    - VLAN
    - MPLS

Libtrace Features

- Consistent and sensible API
  - All the details are handled internally within the library
  - Programmer just focuses on what they need to complete their task
    - Ignore extraneous details
  - Typical libtrace programs require 40% fewer LoC than libpcap

Libtrace Features

- Native support for compressed files
  - Many libpcap tools do not deal with compressed files
    - Most trace files are large and therefore compressed
  - Libtrace doesn't care if a file is compressed or uncompressed
    - Both file types use the same code paths
  - Both gzip and bzip2 are supported

Libtrace Performance

- Threaded I/O
  - All I/O operations are performed in a separate thread
  - Much faster than using a separate gzip process and a pipe
- Avoid memory copies
  - Operate on packets where they arrive, rather than copying into memory allocated by libtrace
- Caching packet details
  - Remembering header locations to avoid decoding whole packet repeatedly
  - Also cache packet lengths and other commonly used data

Getting Libtrace

- Current libtrace version is 3.0.22
  - Available from http://research.wand.net.nz/software/libtrace.php
  - GitHub: http://github.com/wanduow/libtrace
- Open source
  - GPL license
- Operating Systems
  - Linux, FreeBSD, MacOS X
  - Windows is unsupported, but we have built DLLs in the past

Installing Libtrace

- Requirements
  - automake-1.9 or later
  - libpcap-0.8 or later
  - flex and bison

- Strongly recommended
  - zlib-dev (for reading and writing compressed traces)

Installing Libtrace

```
./configure
make
make install
```

- Installs to /usr/local/lib by default
  - Run **`make install`** as root (i.e. using sudo)
  - Append **`--prefix=DIR`** option to **`./configure`** to change

- You may also need to need to:
  - Add /usr/local/lib to /etc/ld.so.conf
  - Run **`sudo ldconfig`**

Libtrace URIs

- URIs describe a libtrace input or output
  - <format>:<location>
  - Example: a PCAP trace file called example.pcap.gz
    - URI is pcapfile:example.pcap.gz

- For input sources, the format can often be left out
  - Libtrace will guess the format by examining the file or interface
  - Format must **always** be specified for outputs

Supported Capture Formats

Format	Base URI	Example	Write
DAG	dag	dag:/dev/dag0	Yes
ERF	erf	erf:/trace/example.erf.gz	Yes
PCAP interface	pcapint	pcapint:eth0	Yes
PCAP file	pcapfile	pcapfile:/trace/example.pcap.gz	Yes
Native Linux (Ring)	ring	ring:eth0	Yes
Native Linux	int	int:eth0	Yes
Native BSD	bpf	bpf:eth0	No
TSH	tsh	tsh:/trace/example.tsh.gz	No
RT protocol	rt	rt:localhost:4500	No
Legacy ATM	legacyatm	legacyatm:/trace/example.atm.gz	No
Legacy Ethernet	legacyeth	legacyeth:/trace/example.eth.gz	No
Legacy PoS	legacypos	legacypos:/trace/example.pos.gz	No
ATM Cell Header	atmhdr	atmhdr:/trace/example.atmhdr.gz	No
FR+	fr+	fr+:/trace/example.fr.gz	No

BPF Filters

- Libtrace supports using BPF expressions to filter traffic
  - Commonly used in PCAP (tcpdump)
  - Many libtrace tools accept BPF as an argument (usually using -f)
    - Tool will then ignore all packets that do not match the filter
  - Be careful when the packets are VLAN-tagged!
    - Precede filter with “vlan and”, otherwise your filter won't match

- Examples
  - “tcp port 80”
  - “vlan and (src host 192.168.2.1 or src host 192.168.2.2)”
  - **`man tcpdump`** for more details

Libtrace Tools

- Libtrace includes a suite of tools for common tasks
  - Splitting, merging traces
  - Dumping packets to a terminal in human-readable form
  - Simple statistical analyses

- Anyone can use the libtrace tools
  - No programming knowledge required
  - You do need to know about URIs and BPF filters
  - Use them to help validate the results from your programs!

Tracepktdump

- Prints the contents of each packet to the terminal
  - Basically the libtrace equivalent of tcpdump
  - The tool I use the most :)
- Decodes headers up to and including the transport layer
  - Prints the name and value for each decoded header field
  - Subsequent payload is dumped as both hex and ASCII
- Will generate a lot of output!
  - Pipe the output through the Unix tool `less`

Tracepktdump

- Usage

```
tracepktdump [-f filter] [-c count] inputURI
```

- Options

  `-f filter` Output only packets matching the BPF expression

  `-c count` Stop after displaying `count` packets

- Example – display all packets on TCP port 80

```
tracepktdump -f "tcp port 80" pcapfile:example.pcap.gz | less
```

Tracesplit

- Divides a trace into subtraces
  - Time interval, e.g. every hour or 5 minutes
  - Number of packets, e.g. every 10,000 packets
  - Size of the output file, e.g. create a series of 1 GB trace files
  - Covering a certain time period, e.g. from 3pm to 5pm
  - Only packets matching a BPF filter, e.g. “tcp port 80”

- Output trace format does not have to match input format
  - tracesplit doubles as a trace format conversion utility!

Tracesplit

- Usage

```
tracesplit [flags] inputURI ... outputURI
```

- Flags

-f filter Only output packets that match this BPF filter

-c count Split every `count` packets

-b bytes Split whenever the output trace reaches `bytes` bytes in size

-i interval Split every `interval` seconds of trace time

-s start Start splitting at this time (UTC seconds)

-e end End splitting at this time (UTC seconds)

-m maxfiles Create a maximum of `maxfiles` files

-z level Sets a compression level for the output traces

-Z type Compress output traces using compression format `type`

-S length Truncate packets to at most `length` bytes

Tracesplit

Create a single trace containing 1000 SMTP packets

```
tracesplit -z 1 -Z gzip -f "tcp port 25" -c 1000 -m 1
pcapfile:input.pcap.gz pcapfile:1000smtp.pcap.gz
```

Split a single long PCAP trace into 1 hour ERF traces

```
tracesplit -i 3600 pcapfile:input.pcap.gz erf:hoursplit
```

Grab a particular 30 minute segment from a trace

```
tracesplit -z 1 -Z gzip -s 1228125600 -e 1228127400
pcapfile:today.pcap.gz pcapfile:interesting.pcap.gz
```

Capture traffic from a live interface into a ERF trace file

```
tracesplit -z 1 -Z bzip2 int:eth0 erf:capture.erf.bz2
```

Libtrace Tools

- tracepktdump – display packets in a human-readable format
- tracesplit – divide a trace into subtraces
- tracemerge – merges traces together
- traceanon – sanitises packets, replacing IP addresses and checksums
- tracestats – performs simple analysis on a given trace
- tracesummary – reports basic statistical summary of a trace
- tracetop – reports busiest network flows in real-time
- tracetopends – reports busiest network endpoints
- tracereport – produces reports describing specific trace properties
- tracertstats – produces real-time packet and byte counts for a trace
- tracereplay – replays a trace to an interface, preserving original timing

To Be Continued....

- Thursday: discuss the programming API
  - I recommend brushing up on your C
  - http://www.wand.net.nz/~salcock/301/
    - Lectures 1-6 will be most relevant for libtrace programming

Developing Libtrace Programs

- The existing tools are handy but don't do everything
  - Need to write our own programs for custom analysis

- Libtrace provides a nice programming API
  - You'll need to develop using C or C++
  - There is also a Python API, but I won't be covering that today

Libtrace Documentation

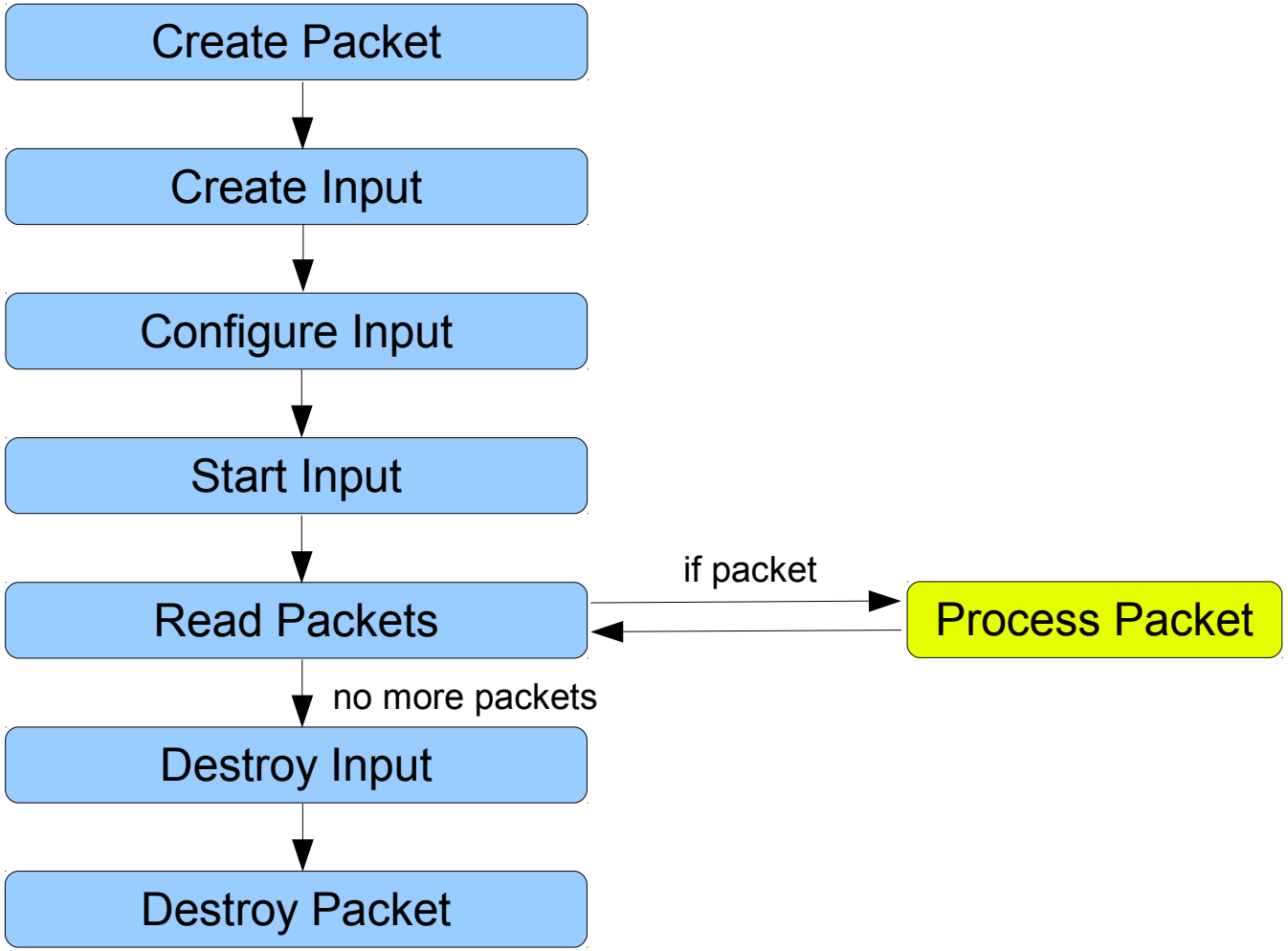
- libtrace.h
  - Libtrace header file, documents all structures and functions
  - Doxygen
    - http://research.wand.net.nz/software/libtrace-docs/html/files.html

- Libtrace Wiki
  - http://github.com/wanduow/libtrace/wiki/

Examples

- All of the examples I'll mention are available online
  - http://www.wand.net.nz/~salcock/tutorial/codedemo
  - Most are included with the libtrace source code as well
    - Look inside the examples/tutorial directory

Basic Program Structure



Libtrace Data Structures

`libtrace_packet_t`

- Represents a single packet within libtrace

`libtrace_t`

- Represents a source of packets within libtrace

`libtrace_out_t`

- Represents an output location within libtrace

Libtrace Data Structures

- The libtrace data structures are *opaque*
  - Don't use or modify the members of these structures
    - Think of them as 'private' members
  - Use the libtrace API functions

Creating Libtrace Packets

```
libtrace_packet_t *trace_create_packet();
```

- Creates a structure for reading packets into
- Returns a pointer to an initialised libtrace packet
- Returns NULL in the event of an error

- Libtrace packets can (and should) be re-used
  - Most applications only need to create one libtrace packet

Creating a Libtrace Input

```
libtrace_t *trace_create(char *uri);
```

- Opens a trace file or live capture for reading
- Location and format specified using the `uri` parameter
- Returns a pointer to the libtrace input structure (libtrace_t)
- Returns NULL if an error occurs
- Created trace is not yet available for reading!

Configuration

```
int trace_config(libtrace_t *trace, trace_option_t option,
  void *value);
```

- Set a configuration option for a trace
- Configuration changes are applied when the trace is started
- Returns -1 if configuration failed, 0 otherwise
- Some possible options for input traces
  - **TRACE_OPTION_SNAPLEN**
  - **TRACE_OPTION_PROMISC**
  - **TRACE_OPTION_FILTER**

Starting

```
int trace_start(libtrace_t *trace);
```

- Prepares a trace file or live capture for reading
- Applies any configuration options
- `trace` must have been previously created using trace_create()
- Returns 0 if successful, -1 if an error occurs
- Packets can now be read from the trace

Error Checking

```
bool trace_is_err(libtrace_t *trace);
```

- Returns true if the error state is set for the given trace
- Does not reset the error state

```
void trace_perror(libtrace_t *trace, const char *msg...);
```

- Very similar to perror() in standard C
- Prints a (hopefully useful) error message to stderr
- `msg` is prepended to the error message
- Clears the error status for the trace

Cleaning Up

```
void trace_destroy(libtrace_t *trace);
```

- Closes a trace and frees up any resources it was using

```
void trace_destroy_packet(libtrace_packet_t *packet);
```

- Fairly self-explanatory
- Frees all resources associated with the packet

Reading Packets

```
int trace_read_packet(libtrace_t *trace, libtrace_packet_t
  *packet);
```

- Reads the next available packet from `trace` into `packet`
- `trace` must have been successfully started
- If `packet` already contains a packet, it will be replaced
- Returns 0 on EOF, -1 on error, otherwise the number of bytes read
  - Remember to handle errors appropriately!

My First Libtrace Program

- Example – readdemo.c
  - A simple program that reads and counts packets
  - Demonstrates basic libtrace program structure

- Building our program and linking against libtrace

```
gcc -g -Wall -o readdemo readdemo.c -ltrace
```

- If libtrace is installed to a non-default location

```
gcc -L/home/salcock/install/lib -I/home/salcock/install/include
  -o readdemo readdemo.c -ltrace
```

Timestamps

```
uint64_t trace_get_erf_timestamp(libtrace_packet_t *packet);

struct timeval trace_get_timeval(libtrace_packet_t *packet);

double trace_get_seconds(libtrace_packet_t *packet);
```

- Returns the time that the provided packet was captured
- If capture timestamp is in a different format, it will be converted
  - e.g. calling `trace_get_timeval` on a ERF trace
- Time formats vary in accuracy and resolution
  - Timestamps are usually measured in seconds since the epoch
  - i.e. Jan 1 1970, 00:00:00 UTC

Timestamps

- Example - timedemo.c
  - Using timestamps to print counts every 10 seconds of trace time

Packet Length

- Capture length
  - The current size of the packet
  - Does not include the capture format header
  - If the packet has been truncated, this is how much of the packet is available
- Wire length
  - The size of the packet when it was first captured, i.e. before any truncation
  - Does not include the capture format header
  - Can include the Frame Check Sequence on Ethernet packets
    - DAG captures retain FCS, PCAP does not

Packet Length

- Payload length
  - The amount of post-header payload that was in the packet
    - i.e. the payload after the TCP/UDP/ICMP header
  - Value is based on the original packet before truncation
  - Useful for studying application behaviour, e.g. HTTP, DNS

- Framing length
  - The size of the capture format header
  - Unlikely that you will ever need to know the framing length

Packet Length

```
size_t trace_get_capture_length(libtrace_packet_t *packet);

size_t trace_get_wire_length(libtrace_packet_t *packet);

size_t trace_get_payload_length(libtrace_packet_t *packet);

size_t trace_get_framing_length(libtrace_packet_t *packet);


size_t trace_set_capture_length(libtrace_packet_t *packet,
  size_t size);
```

- Truncates the packet to the suggested length
- If `size` is larger than the current capture length, the packet is unchanged
- Returns the new capture length

Packet Length

- Example - lengthdemo.c
  - Instead of just counting packets, let's try counting bytes

Endpoint Functions

```
uint16_t trace_get_source_port(libtrace_packet_t *packet);

uint16_t trace_get_destination_port(libtrace_packet_t *packet);
```

- Returns the requested port number from the transport header
- The port number is returned in HOST byte order
- Returns 0 if no port number is available

Endpoint Functions

```
struct sockaddr *trace_get_source_address(libtrace_packet_t
  *packet, struct sockaddr *addr);

struct sockaddr *trace_get_destination_address(libtrace_packet_t
  *packet, struct sockaddr *addr);
```

- Returns the requested IP address as a sockaddr
- If `addr` is NULL, static storage is used to store the result
- Returns NULL, if no IP address is present (i.e. not an IP packet)
- Works for v4 or v6
- Some knowledge of sockaddr conventions in C is required

Endpoint Functions

```
char *trace_get_source_address_string(libtrace_packet_t *packet,
char *space, int spacelen);

char *trace_get_destination_address_string(libtrace_packet_t
*packet, char *space, int spacelen);
```

- Returns the requested IP address as a string
  - Works for both IPv4 and IPv6
- `space` should point to a character buffer for storing the result
- `spacelen` must be set to the size of the `space` buffer
- If `space` is NULL, static storage is used to store the result
  - Use it or lose it!
  - Set `spacelen` to 0 if space is NULL

Endpoint Functions

```
uint8_t *trace_get_source_mac(libtrace_packet_t *packet);

uint8_t *trace_get_destination_mac(libtrace_packet_t *packet);
```

- Returns a pointer to the requested MAC address
- Works for both Ethernet and 802.11 frames
- Returns NULL if no MAC address available

Endpoint Functions

- Example – sourcedemo.c
  - Print the source MAC address, IP address and port for each packet
  - Demonstrates how to use sockaddrs correctly

Protocol Decoding

- Protocol decoders fall into three categories
  - Layer access – jump to first header at a given OSI layer
  - Get payload – skip past a given header
  - Shortcut – jump straight to a specific header
    - Easy to use, but less flexible
    - Don't use these if you want to use a 'get payload' decoder

Layer Access Functions

- Find the header at a specific OSI layer
  - **`trace_get_packet_buffer`**
    - Returns the very start of the packet (post-capture format header)
  - **`trace_get_packet_meta`**
    - Returns the first meta-data header, e.g. RadioTap
  - **`trace_get_layer2`**
    - Returns the first link layer header, e.g. Ethernet, 802.11
  - **`trace_get_layer3`**
    - Returns the IP layer header, e.g. IPv4, IPv6
  - **`trace_get_transport`**
    - Returns the transport header, e.g. TCP, UDP, ICMP

Layer Access Functions

```
void *trace_get_transport(libtrace_packet_t *packet,
   uint8_t *proto, uint32_t *remaining);
```

- proto and remaining are used as 'output' variables
  - proto will tell you which transport protocol is used
  - remaining tell you how much captured data is remaining in the packet, starting from the returned header
- Returns the address of the transport header
  - If NULL, no transport header was present in the packet
  - Based on the value of proto, you must cast this to the right type
  - Can return an incomplete header – check remaining!

Examining Headers

- Libtrace defines structures for each common header
  - Header fields are stored in NETWORK byte order
  - Header structures are defined in libtrace.h
  - Optional fields are NOT included in the structure definition
  - Example: Libtrace UDP header structure

```
/** Generic UDP header structure */
typedef struct libtrace_udp {
  uint16_t      source;       /**< Source port */
  uint16_t      dest;         /**< Destination port */
  uint16_t      len;          /**< Length */
  uint16_t      check;        /**< Checksum */
} PACKED libtrace_udp_t;
```

Byte Ordering

- Byte ordering bugs are common in libtrace programs
  - Network byte order: big-endian
  - Host byte order: little-endian (usually, depends on architecture)

- Packet contents are in network byte order
  - This is a consequence of the zero-copy approach
  - Must convert values to host byte order before using them
  - Some API functions do the conversion automatically
    - e.g. `trace_get_source_port`

Byte Ordering

- Avoiding byte ordering bugs
  - Byte ordering is only a problem for fields > 1 byte in size
    - So uint8_t is ok, uint16_t and uint32_t are at risk
  - If reading directly from a libtrace header structure, byteswap
  - If an API function returns a value directly, don't byteswap
  - Use ntohs to byteswap 16 bit values, ntohl for 32 bit values
  - Above all, double check your results
    - If they don't make sense, maybe you've got a bug

Protocol Decoding

- Example – gettransportdemo.c
  - Using trace_get_transport to examine UDP traffic

Get Payload Functions

- Step past a single header in a packet
  - Very useful when there are multiple headers at a single layer
    - e.g. decoding VLAN tags
  - Only way to access application-layer payload
  - Skips over any options appended to the header, e.g. TCP options
  - To use these, you must pass in a valid value for 'remaining'
    - Only can get this from a layer access function

Get Payload Functions

```
void *trace_get_payload_from_ip(libtrace_ip_t *ip,
   uint8_t *proto, uint32_t *remaining);
```

- Again, proto and remaining are used as output variables
- However, remaining MUST also contain the value resulting from the function call that returned the IP header
  - In this case, that would be `trace_get_layer3`
- Returns a pointer to the start of the header following the IP header
  - Returns NULL if there was no subsequent header
  - Does NOT return NULL if the header was truncated
    - Again, be careful to check the value of remaining

Get Payload Functions

- Example – getpayloaddemo.c
  - Using get_payload_from_ip to find the UDP header instead

Shortcut Functions

- Simpler versions of the layer access functions
  - Request the header for a specific protocol
    - Only a handful of protocols have shortcut functions
  - Returns NULL if there is no matching COMPLETE header
  - Limitation: no remaining value is returned
    - This means you cannot follow up with a get payload function

```
libtrace_ip_t *trace_get_ip(libtrace_packet_t *packet);

libtrace_ip6_t *trace_get_ip6(libtrace_packet_t *packet);

libtrace_tcp_t *trace_get_tcp(libtrace_packet_t *packet);

libtrace_udp_t *trace_get_udp(libtrace_packet_t *packet);

libtrace_icmp_t *trace_get_icmp(libtrace_packet_t *packet);
```

Writing Packets

- The output API is very similar to the input API
  - Just add an extra 'output' or 'out' to everything :)

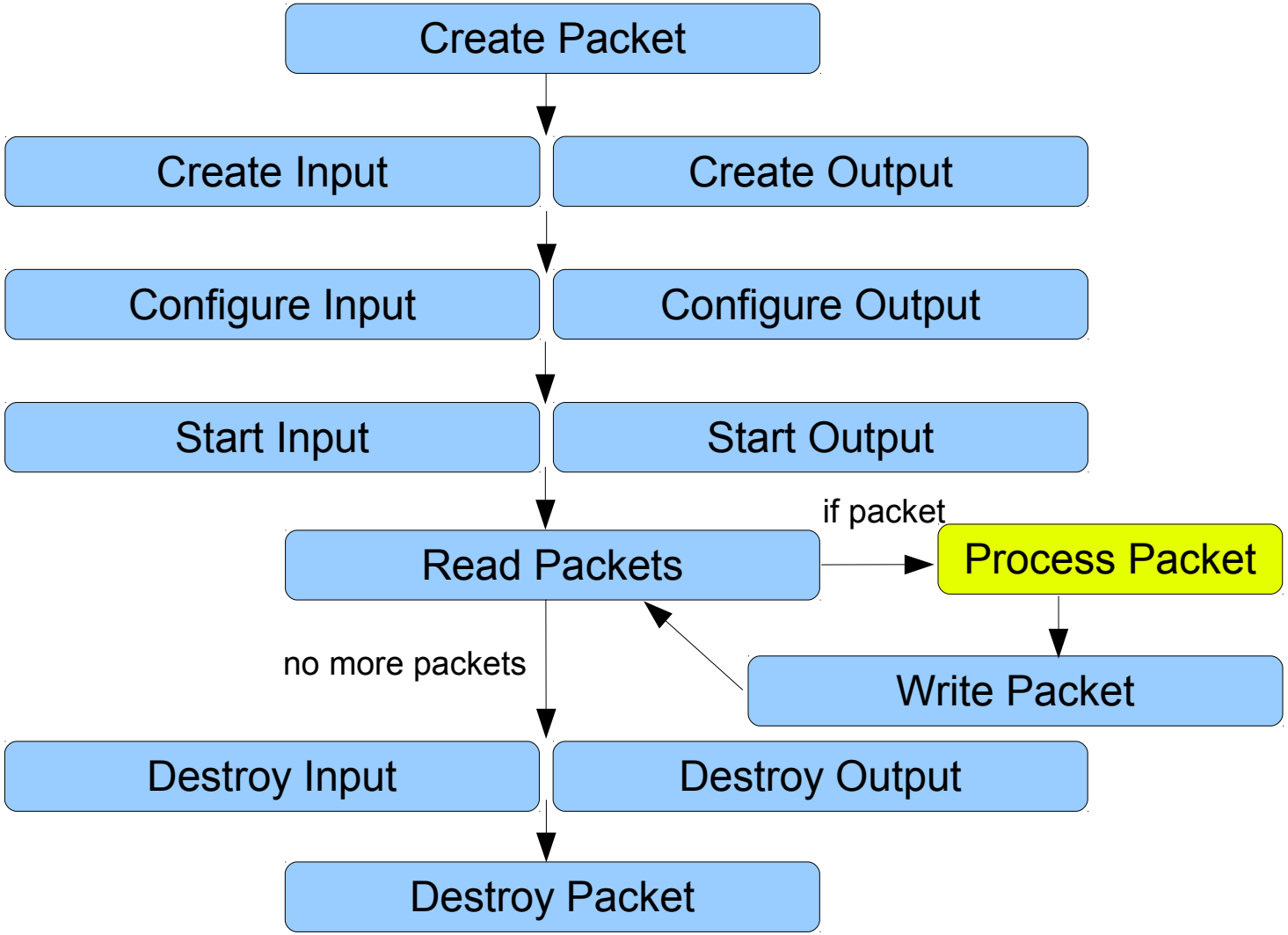
```
libtrace_out_t *trace_create_output(char *uri);

int trace_config_output(libtrace_out_t *trace,
  trace_option_output_t option, void *value);

int trace_start_output(libtrace_out_t *trace);

void trace_destroy_output(libtrace_out_t *trace);

bool trace_is_err_output(libtrace_out_t *trace);

void trace_perror_output(libtrace_out_t *trace, const char
  *msg...);
```

Writing Packets

```
int trace_write_packet(libtrace_out_t *trace,
  libtrace_packet_t *packet);
```

- Writes the given packet to the output trace
- Returns -1 if an error occurs, otherwise the number of bytes written

Basic Program Structure with Writing



Writing Packets

- Example - writedemo.c
  - Create a trace containing only TCP port 25 traffic

Writing Packets

- By default, output traces are not compressed
  - This can be changed during the configuration step
  - **`TRACE_OPTION_OUTPUT_COMPRESS`** sets the compression level
    - 1 = least compression, 9 = max compression
  - **`TRACE_OPTION_OUTPUT_COMPRESSTYPE`** sets a compression method
    - “gzip”, “bzip2” are valid methods

Filtering Packets

```
libtrace_filter_t *trace_create_filter(char *filterstring);
```

- Creates a libtrace filter object
- The filter string should be expressed using BPF
  - e.g, “tcp port 80”
- Will always return a valid filter – not compiled until first applied

```
void trace_destroy_filter(libtrace_filter_t *filter);
```

- Deallocates all resources associated with a libtrace filter

Filtering Packets

- Applying a filter at the configuration step
  - Best way to apply a filter to an entire input
  - **`trace_read_packet`** will only return packets that match the filter

```
libtrace_filter_t *filt = trace_create_filter("tcp port 80");

if (trace_config(trace, TRACE_OPTION_FILTER, filt) == -1) {
        trace_perror(trace, "Failed to apply filter");
        trace_destroy_filter(filt);
        return -1;
}

/* TODO Start trace, read some packets, etc. */

trace_destroy_filter(filt);
```

Filtering Packets

```
int trace_apply_filter(libtrace_filter_t *filter,
  libtrace_packet_t *packet);
```

- Applies a libtrace filter to an individual packet
- Returns 0 if the filter does not match, >0 if it does
- Returns -1 if an error occurs

Filtering Packets

- Examples
  - filterdemo.c – a simple version of tracefilter
    - Uses trace_apply_filter
  - configdemo.c – a better version of tracefilter
    - Uses trace_config to apply a filter
    - Also uses trace_config_output to compress the output trace

Additional Reading

- A paper on libtrace
  - Beautifully written description of the library and how it works
  - Includes performance test results
  - http://www.sigcomm.org/ccr/papers/2012/April/2185376.2185382
  - Or just Google it :)

The Assignment

- Due 5pm, Friday April 1

- 4 Tasks
  - Basic trace analysis
  - Identifying port scanning
  - Determining the cause of a network event
  - Short answer questions

- Libtrace programming in C required
  - Better brush up on your C

Assignment Hints

- Do everything you can to demonstrate understanding
  - Use the examples, but don't copy them directly
  - Document what is going on in your own words

- This is an INDIVIDUAL assignment
  - Discussion with others is OK
  - Don't loan your code to your classmates!

Assignment Hints

- Think like a researcher
  - Validate your results using the existing tools
  - Think about whether the results you get make sense
    - e.g. what ports do you expect to see the most traffic on?

- Code like a programmer
  - Check for errors
  - Indent sensibly
  - Avoid memory leaks
  - Document

Assignment Hints

- Common mistakes
  - Byte ordering bugs – check your results!
  - Failure to read instructions carefully
    - Output formatting and configuration
  - Copying too much from the examples
    - I will deduct marks for this!
  - Copying from your classmates
  - Failing to communicate effectively

Assignment Help

- Moodle discussion forum
  - I'll be monitoring the forum
- Make an appointment to see me
  - Email salcock@waikato.ac.nz
  - Don't leave it until the last minute!