How to (passively) understand the application layer?

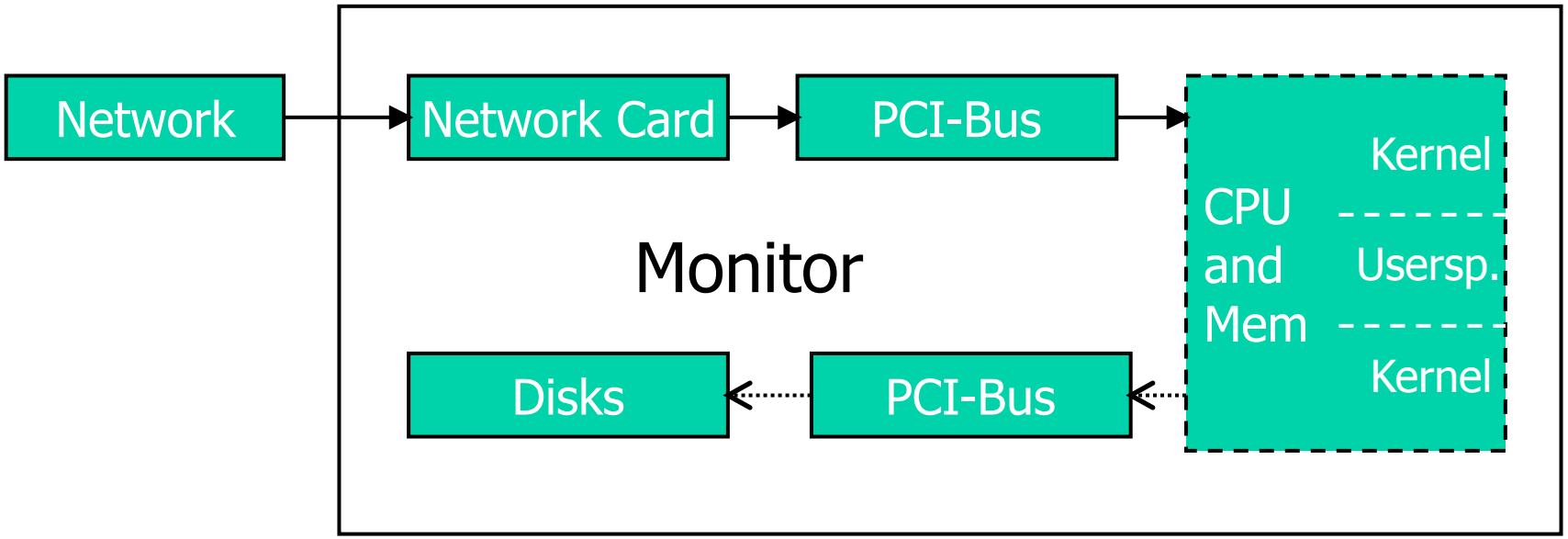
Packet Monitoring

What to expect?

- Overview / What is packet monitoring?
- How to acquire the data
- Handling performance bottlenecks
- Analyzing the transport and application layer
- (Mis-)Using the Bro Network Intrusion Detection System (NIDS) for network measurements

What is a packet monitor?

- Measuring / recording network data on a per packet basis
- Ordinary (although high-end) PC hardware
- Datapath:



Passive Monitoring: Challenges (1)

- User privacy & network security
- Data privacy vs. data sharing
- Data filtering
- Tap into live network traffic and extract packets
- Must not interfere with normal packet transmission
- Real-time: cannot control bandwidth, cannot postpone work

Passive Monitoring: Challenges (2) Performance Issues

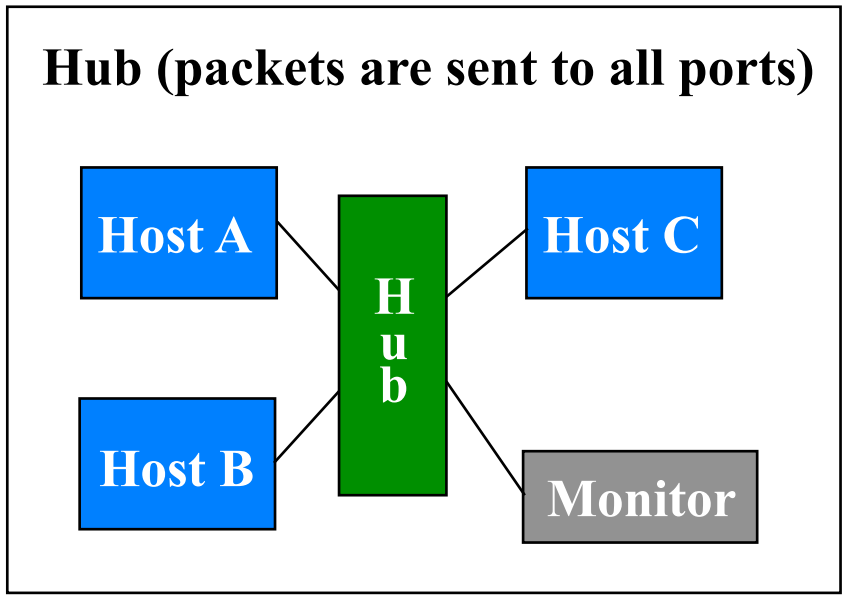
- High data rate
  - Bandwidth limits on CPU, I/O, memory, and disk/tape
  - Network cards optimized for bi-directional data transfer, not capturing
- High data volume
  - Space limitations in main memory and on disk/tape
  - Could do online analysis to sample, filter, & aggregate
- High processing load
  - CPU/memory limits for extracting, counting, & analyzing
  - Could do offline processing for time-consuming analysis
- General solutions to system constraints
  - Sub-select the traffic (addresses/ports, first *n* bytes)
  - Kernel and interface card support for measurement

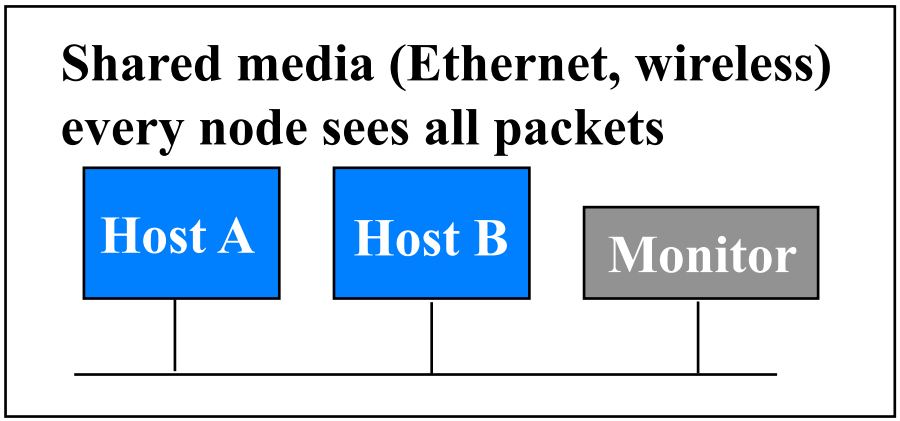
Monitoring Links: Overview

- How to get data off the network, without interfering normal transmission?
- For half-duplex:
  - Shared medium
  - Hub
- For full-duplex:
  - Monitor / SPAN port
  - "Tap"

Monitoring Links (1)

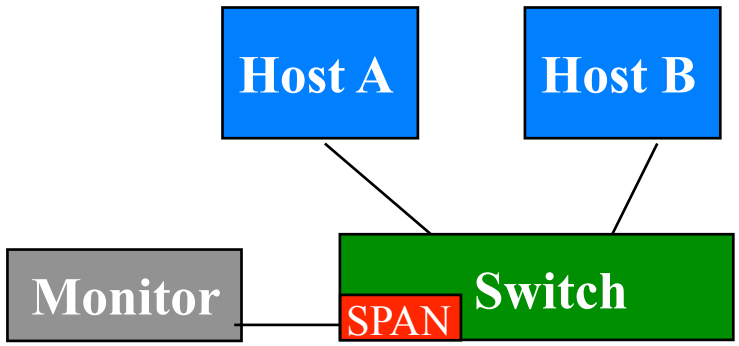
- Half Duplex: host cannot send and receive at the same time. Only one host can send.





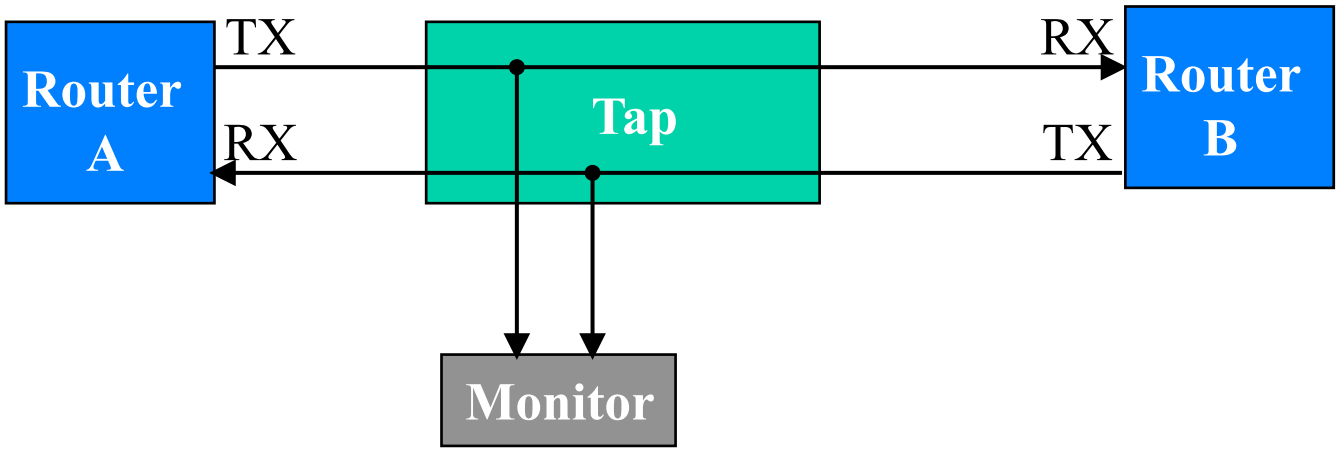
Monitoring Links (2)

- Full Duplex: host can send and receive at the same time
- Monitoring- / SPAN port on switch
  - every packet seen by switch copied to SPAN port
  - easy (every switch supports this)
  - all sending host are aggregated into one monitoring link ==> Packet loss



Monitoring Links (3)

- Full Duplex
- Tap into data
  - Only between two nodes (routers)
  - Can capture all traffic
  - Need two receive ports on Monitor
  - Fiber: purely optical
  - Copper: needs active components



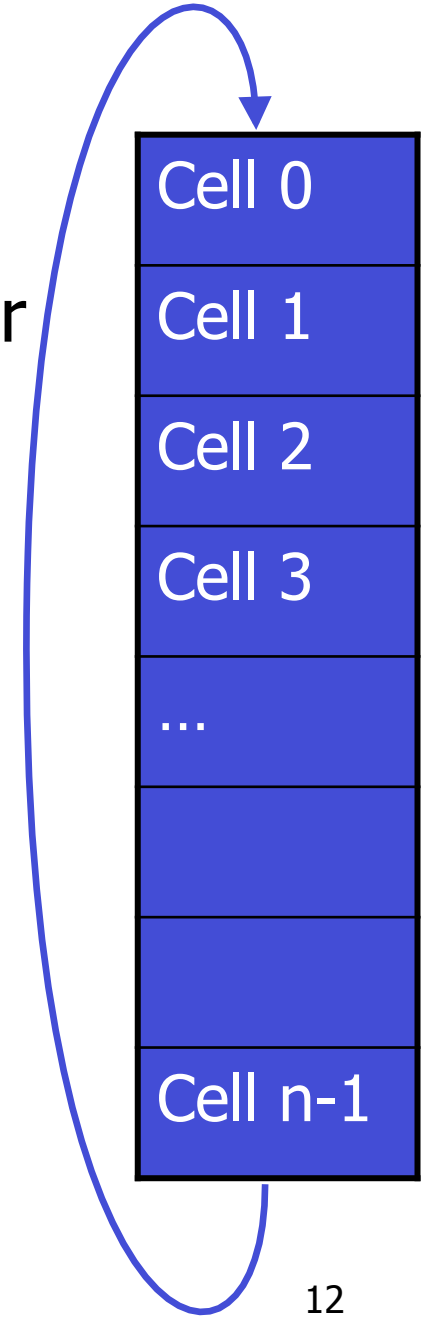
Handling Performance Bottlenecks

Handling high bandwidth

- Hard for monitors to cope with high load
  - Interrupt VS Polling
  - Long datapath: data copied several times => Ring buffer
  - Timestamping?
- Dedicated network monitoring cards
  - Often several ports (for Taps)
  - Filtering / aggregation in hardware on card
  - Very expensive (EUR 3,500 for 1G, EUR 20k for 10G)
- Split or aggregate traffic utilizing off-the-shelf switching hardware

Ring buffers

- Physical chunk of RAM provided by driver
  - Organized as logical ring
  - Mapped into user-space when capturing
- Buffer handling: write pointer advanced
  - By driver if ordinary network card or
  - In hardware by monitoring card, e.g., DAG
- User-space process captures data
  - Advances read pointer
  - Writes to disk or live analysis
- Use DMA if possible!



Analyzing the transport and application layer

How to get from packets to
connections (TCP, UDP) to
application level protocols (HTTP, DNS, etc.)

Packet VS Connections VS Applications

- Monitors deliver single packets
- Lots of measurements one can do on per packet basis
  - Timing, packet sizes, routing, IP stats, ...
- More measurements on transport layer (TCP/UDP)
  - Timing, connection size, ...
- But often we want to analyze application layer protocols (e.g., HTTP, SIP, etc.)

Application-level Messages

- Application-level transfer may span multiple packets
  - Demultiplex packets into separate “flows”
  - Identify by source/dest IP addresses, ports, and protocol
  - Maybe also application level identifiers

Application-level Messages: Reassembly

- Reconstructing ordered, reliable byte stream
  - De-fragment fragmented IP packets
  - Reassemble TCP segments
  - Sequence number and segment length in TCP header
  - Buffer to deliver packets in correct order to analyzer
  - Drop duplicates
- Packets might be missing (measurement drops)
- Packet might be truncated

Application-level Messages: Reassembly (2)

- Inconsistent retransmissions
  - TCP retransmission, but data does not match
- Need state per connection
  - How many connections?
  - How do you handle attacks, e.g., network scans?
- Idle connections
  - Is teardown missing?
  - Is there going to be more data?
  - Cannot keep state forever (memory exhaustion)
  - => need strategy for state removal

Application-level Messages

Extraction of application-level messages

- Parsing the syntax of the application-level
  - Clients/Servers may not adhere to specification
- Identifying the start of the next message,

  e.g., HTTP
  - Absence of body
  - Presence of Content-Length
  - Chunk-encoded message
  - Multipart/byterange
  - End of TCP connection

(Mis-)Using the Bro Intrusion Detection System for Network Measurement

What is a NIDS?

- Network Intrusion Detection System (NIDS)
  - Monitors network traffic to detect attacks in real-time
  - Reports suspicious activity to operator
- A NIDS has to be robust
  - To protect itself from direct attacks
  - To detect / prevent evasions
  - Must be careful to not exhaust its resources (memory, CPU, disk)

Why use a NIDS for measurement?

- To perform its task a NIDS has to
  - De-Fragment IP packets
  - Reassemble TCP connections (and cope with inconsistencies and packet loss)
  - Keep track of connections, manage state
  - Track resource usage
  - Parse Application-layer protocols
  - Extract Application-layer messages and data elements
    - e.g., URLs, etc.
  - Handle broken protocol implementations
- All this things are also relevant to measurement

Bro: an open source NIDS

- Bro is open source
- Developed by Vern Paxson (UC Berkeley)
- Used as productive and research system
- Is modular and easy to extend
- We use it heavily
  - As NIDS to protect our network
  - Conduct NIDS research
  - Conduct network measurements

Bro System Philosophy

- Fundamentally, Bro provides a real-time network analysis framework
- Emphasis on
  - Application-level semantics
    - rare to analyze individual packets
  - Tracking information over time
    - Both within and across connections
    - Also archiving for later off-line analysis
- Strong separation of mechanism and policy
  - Much of the system is policy-neutral. I.e., no presumption of "good" or "bad"

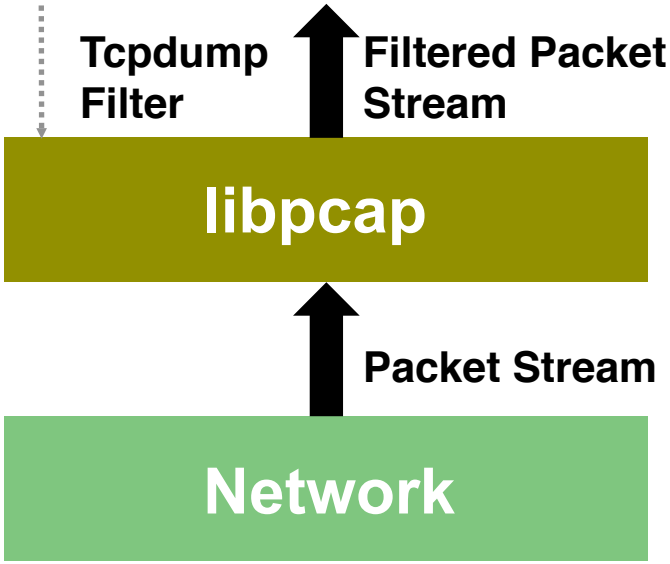
Bro features

- Full TCP stream reassembly
- Stateful protocol analysis
- Dynamic Protocol Analysis (DPD)
- BinPAC – a network protocol description language
- Very flexible policy language (called Bro as well)
  - Specialized for traffic analysis
  - Strongly typed for robustness (conn_id, addr, port, time, …)
  - Can trigger alarms and/or program execution
  - Supports dynamic timeouts
- Clustering support for analysis of multi Gbps links
- Cooperates with Network Time Machine

Inside Bro

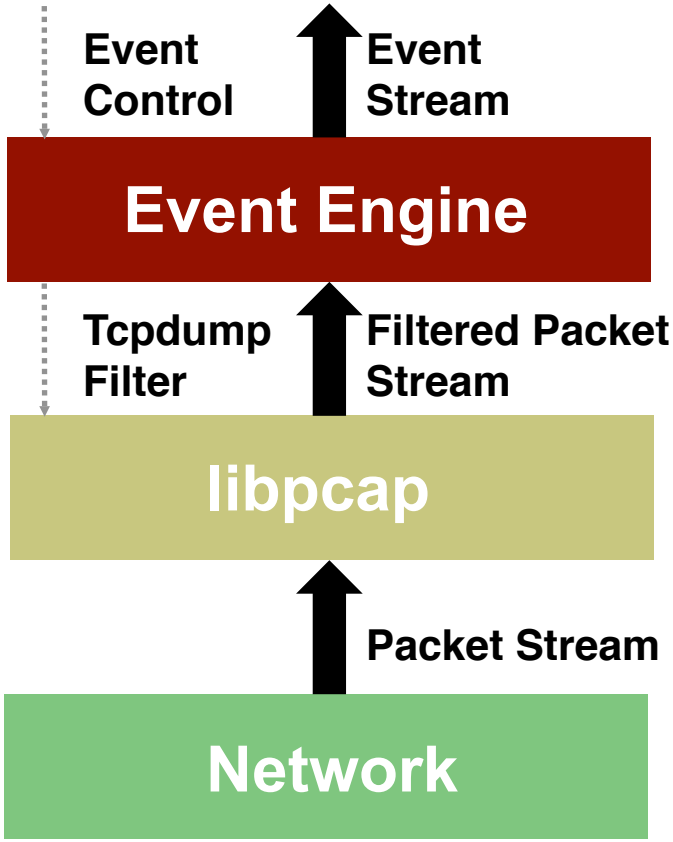
- Passive link tap copies all traffic

Inside Bro



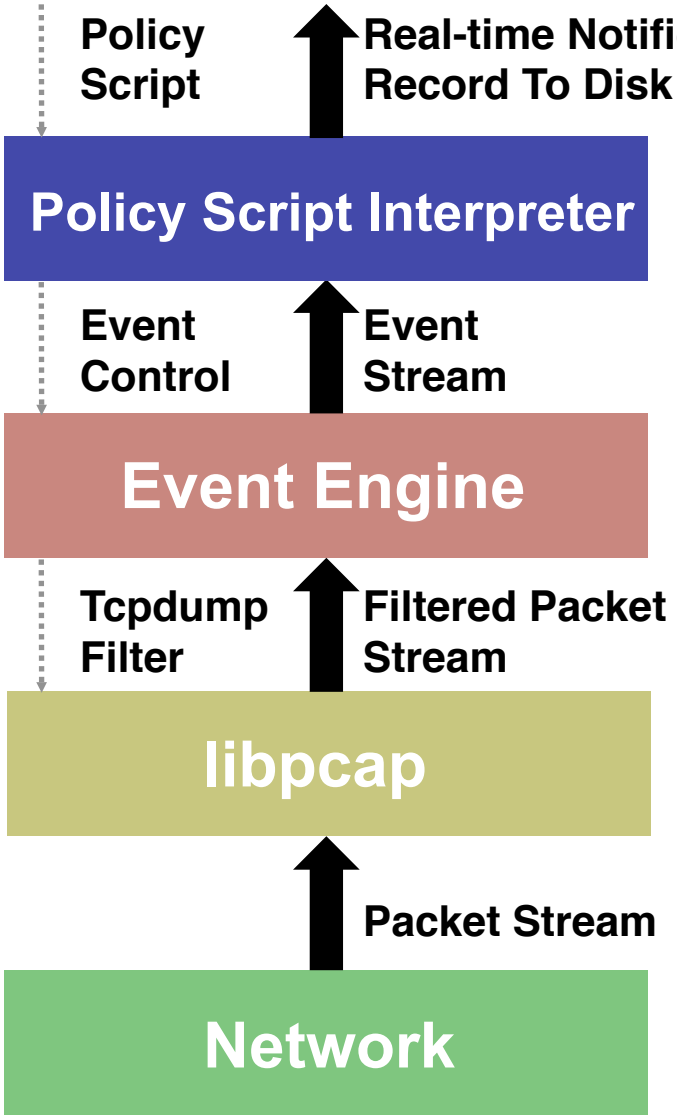
- Kernel filters high-volume stream

Inside Bro



- "Event engine" produces *policy-neutral* events, e.g.:
  - Connection-level:
    - connection attempt
    - connection finished
  - Application-level:
    - ftp request
    - http_reply
  - Activity-level:
    - login success

Inside Bro

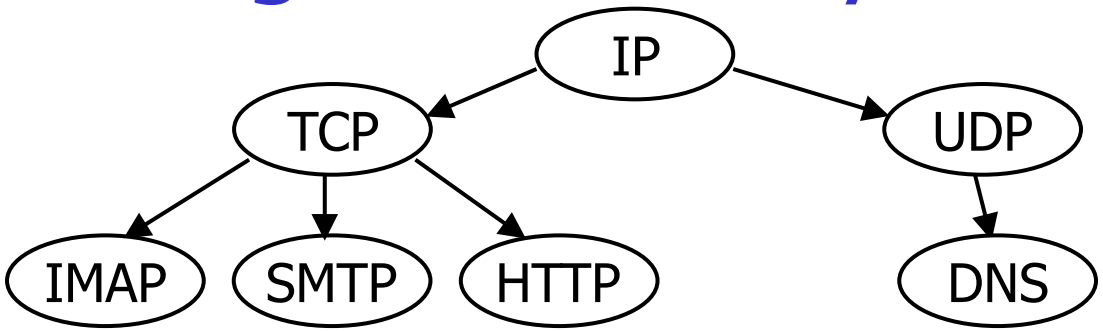


- "Policy script" incorporates:
  - Context from past events
  - Site's particular policies
- ... and *takes action*:
  - Records to disk
  - Generates alerts
  - Executes programs as response

Event Engine

- Event engine performs generic analysis
- Also termed the "Core"
- Written in C++
- Basic element of analysis is a "connection"
  - De-fragment IP
  - Reassemble TCP streams
  - Pass reassembled TCP (UDP) streams to application-level analyzers
  - Event engine uses an analyzer tree

Event Engine – Analzyer tree



- Tree elements can tune-out if not their protocol
- Data transforms as it flows through analyzers
  - E.g., packets -> byte stream -> lines of text
- As analyzers observe activity, they generate events
  - Events span several aggregation levels
  - All events triggered by a given packet executed before next packet is processed

Bro's protocol analyzers

- Full analysis
  - HTTP, FTP, telnet, rlogin, rsh, RPC, DCE/RPC, DNS, Windows Domain Service, SMTP, IRC, POP3, NTP, ARP, ICMP, Finger, Ident, Gnutella, BitTorrent, NNTP, …
- Partial analysis
  - NFS, SMB, NCP, SSH, SSL, IPv6, TFTP, AIM, Skype, …
- In progress
  - BGP, DHCP, Windows RPC, SMB, NetBIOS, NCP, …
- Data sources
  - DAG, libpcap, NetFlow

How to write an analyzer (1)

- Derive from TCP_ApplicationAnalyzer
- Implement interface methods
  - Constructor & Destructor
  - **`DeliverStream`** // Receives new in-order data
  - **`Undelivered`** // called if packet drop detected
  - **`Done`** // removed from Connection table
  - Other methods as needed: **`EndpointEOF`**, **`Connection{Closed,Finished,Reset}`**, …
- For UDP: directly derive from Analyzer, implement **`DeliverPacket`** rather than **`DeliverStream`**

How to write an analyzer (2)

- Make sure your analyzer is used
  - Register it in **`Analyzer::analyzer_configs[]`**
  - Trigger analyzer, e.g., via DPD signatures:

```
signature dpd_bittorrent_peer1 {
  ip-proto == tcp
  payload /^\x13BitTorrent protocol/
  tcp-state originator
}
signature dpd_bittorrent_peer2 {
  ip-proto == tcp
  payload /^\x13BitTorrent protocol/
  tcp-state responder
  requires-reverse-signature dpd_bittorrent_peer1
  enable "bittorrent"
}
```

How to write an analyzer (3)

- Use BinBAC if possible:

```
type BitTorrent_Request = record {
        index:  uint32;
        begin:  uint32;
        length: uint32;
} &let {
        deliver: bool = $context.flow.deliver_request(index,
                                                      begin, length);
};
```

- Throw events
  - Register in event.bif
    - Makes event available in policy layer
    - Gives method for throwing for free
  - Events expensive: only throw if handler registered!

Advanced analyzer features and tricks

- Support analyzers
  - Preprocess data in an analyzer
  - E.g., ContentLine_Analyzer
- Jump over missed packets
- Register expected connections in DPD
  - E.g., FTP data, BT peer traffic after tracker request, …

Policy Script Layer

- Bro specific scripting language
  - Procedural
  - Strong support for network data types (IPs, subnets, ports, etc.)
  - Provides support for state management
- Receives (and processes) events from Event Engine
- Can also generate further events, that are handled by other policy scripts
- Tradeoffs: where to implement functionality
  - Event Engine is fast; Script Layer is easy to implement
  - Event Engine processes protocols, Script Layer implements policy

Native functions

- Scripting relies on native implementation of some functions
  - Basic functionality, e.g, **`fmt()`**
  - Faster execution, e.g., **`cat_string_array()`**
  - Event declaration
- Use **`bifcl`** to implement

```
function cat_string_array%(a: string_array%): string
        %{
        TableVal* tbl = a->AsTableVal();
        return new StringVal(cat_string_array_n(tbl, 1,
                                        a->AsTable()->Length()));
        %}
```

Scripting basics

- Types:
  - **`bool`, `int`, `count`, `enum`, `double`, `enum`, `string`, `addr`, `port`, `net`, `subnet`, `time`, `interval`**
  - Custom types via **`record`**
  - **`table`** and **`set`**
  - Regular expressions
- Types derived from assignment
- Scope: global, const, local
- Attributes: &redef, &default, &optional, …
- Functions and event handlers

Policy state management

- Often policy is used to combine different events
  - => requires state
- Remove state on event connection_state_remove
- Tables and sets have expire functions
  - From netflow.bro:

```
global flows: table[conn_id] of flow
       &write_expire = 31 min
       &expire_func = print_flow;
```

Bro policy example

```
global ssh_log = open_log_file("ssh") &redef;
global did_ssh_version: table[addr, bool] of count
                        &default = 0 &read_expire = 7 days;

event ssh_client_version(c: connection, version: string)
   {
   if ( ++did_ssh_version[c$id$orig_h, T] == 1 )
        print ssh_log, fmt("%s %s \"%s\"", c$id$orig_h, "C",
                                version);

   skip_further_processing(c$id);
   }

event ssh_server_version(c: connection, version: string)
   {
   if ( ++did_ssh_version[c$id$resp_h, F] == 1 )
        print ssh_log, fmt("%s %s \"%s\"", c$id$resp_h, "S",
                                version);
   }
```

Advanced scripting

- Schedule actions into the future: **`schedule`**
  - Starts timer to execute a function
- Asynchronous operation: **`when`** statement
  - Delays execution of current handler until a condition is met
- Keep state over invocations: **`&persistent`**

Bro in measurement practice

- Often: run offline for repeatability
- If live, use traffic splitter to utilize multiple CPU cores
- Use policy to dump relevant data, analyze log with Bash/Python/AWK/Perl script

Advanced Bro Features

- Broccoli = Bro Client Communications Library
  - C interface for external programs to transmit & receive values and events
- Support for external analyses
  - Antivir, libmagic, GeoIP, passive OS fingerprinting...
- Cluster Bro
  - Distribute load across multiple machines
  - Merge event stream to central controller
- Network traffic time machine
  - selective replay of past connections for further analysis
  - Cutoff for extended traffic storage