Intrusion Detection and Malware Analysis
Network IDS sensors

Pavel Laskov
Wilhelm Schickard Institute for Computer Science
A network IDS for this class

- **Detection:**
  - signature matching
  - signature generation

- **Feature extraction:**
  - computation of n-gram embedding
  - token extraction

- **Audit (sniffer):**
  - packet capture
  - packet de-fragmentation
  - TCP stream re-assembly

![Diagram of network IDS components]

- Security policies
- Meta-classification
- Event classification
- Feature extraction
- Audit data
- Alarms
- Events
- Classifier setup
- Filter setup
Processing/sniffing of network data

User space

Applications

Sniffer

Kernel space

TCP stack

NIC driver

Hardware

packets
General sniffer issues

- Do we want to see all traffic or only traffic designated for our host (promiscuous mode or not)?
- Do we want to see traffic for all services or only a selected subset thereof?
- Traffic granularity:
  - Layer 2 frames
  - Layer 3 packets
  - Layer 4 connections / flows
  - Layer 7 events (requests, sessions)
Libpcap: packet capture library

- Initially developed in 1994 for Tcpdump utility at LBNL (BSD license)
- Windows port WinPcap
- Used by a large number of security-related products:
  - Tcpdump
  - Wireshark
  - Snort
  - Bro
  - nmap
  - NetworkMiner
- Written in C, wrappers available for various languages: Perl, Python, Java, Ruby, Tcl, Lisp, .NET
Main libpcap application steps

- Determine an interface to sniff on.
- Open a device for sniffing, set up a sniffing mode (promiscous or not).
- Define a filter expression and compile it into an internal pcap representation.
- Enter a packet capture loop.
- Close a session and cleanup.
Setting up a device

- Explicitly specifying an interface as a string:
  ```c
  int main(int argc, char *argv[]) {
    char *dev = argv[1];
    printf("Device: %s\n", dev);
  }
  ```

- Looking up for a suitable device:
  ```c
  int main(int argc, char *argv[]) {
    char *dev; errbuf[PCAP_ERRBUF_SIZE];
    dev = pcap_lookupdev(errbuf);
    if (dev == NULL) {
      fprintf(stderr, "Errmsg: %s\n", errbuf);
    }
    printf("Device: %s\n", dev);
  }
  ```

- For multiple interfaces, use
  ```c
  int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
  ```
Opening a device for sniffing

```c
pcap_t *pcap_open_live(char *device, int snaplen,
                        int promisc, int to_ms, char *ebuf)
```

Arguments:
- `*device`: device name to be opened
- `snaplen`: number of bytes to be captured
- `promisc`: promiscuous mode or not
- `to_ms`: read timeout in milliseconds
- `*ebuf`: error buffer

Return value:
- `*pcap_t`: pcap handle pointer
Berkeley packet filter (BPF): an extensive language for specifying packet filtering rules

Examples:

```
tcpdump host helios and !( hot or ace )
tcpdump 'gateway snup and ip[2:2] > 576'
tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0'
```

Compiling a BPF:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int opt bpf_u_int32 netmask)
```

Setting up a BPF:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```
Getting a single packet off the wire:

`u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)`

- Packet content is returned as `u_char*` (byte array)
- Packet info is stored in the appropriate structure (not to be confused with a packet header!):

```c
struct pcap_pkthdr {
    struct timeval ts; /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len; /* length this packet (off wire) */
};
```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

- Reads cnt packets off the wire (a negative value indicates an infinite loop) and calls callback for each packet.
- Arbitrary user input can be passed in a byte array *user.
void got_packet(u_char *args, const struct pcap_pkthdr *header,
               const u_char *packet);

- Packet content is stored in u_char* packet.
- Packet info is stored in the appropriate structure.
- All the processing of packet content/info is encapsulated in a callback.
Making sense of packet content

```c
#define SIZE_ETHERNET 14
const struct sniff_ethernet *ethernet; /* The ethernet header */
const struct sniff_ip *ip; /* The IP header */
const struct sniff_tcp *tcp; /* The TCP header */
const char *payload; /* Packet payload */

u_int size_ip;
u_int size_tcp;

ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
```
Beyond packet capture

- How do we handle packet fragments?
- How do we assemble packets belonging to the same TCP connections?
Libnids: network IDS library

- **Main features:**
  - Packet capture and replay (via libpcap)
  - Packet de-fragmentation
  - TCP stream reassembly
  - API via user-defined callbacks

- **Usage example:**

  ```c
  void tcp_callback(struct tcp_stream *tcp, void ** ptr) {
    // Define a user callback for TCP connections
  }

  main() {
    nids_params.device = "ppp0";
    if (!nids_init()) exit(1);
    nids_register_tcp(tcp_callback);
    nids_run();
  }
  ```
Initialization is controlled by a global variable

```c
struct nids_prm nids_params
```

Most important fields:

- `char *device`: network interface
- `char *filename`: file name for offline processing
- `char *pcap_filter`: pcap (tcpdump!) filter string
- `int promisc`: you known what you are looking for!
libnids callbacks

- **IP fragment callback**: invoked upon arrival of each IP fragment
  
  ```c
  void ip_frag_callback(struct ip* packet, int len)
  ```

- **IP packet callback**: invoked upon arrival of each IP packet
  
  ```c
  void ip_callback(struct ip* packet, int len)
  ```

- **TCP callback**: invoked upon arrival of each TCP packet
  
  ```c
  void tcp_callback(struct tcp_stream *ns, void **param)
  ```

- **UDP callback**: invoked upon arrival of each UDP packet
  
  ```c
  void udp_callback(struct tuple4 *addr, char *buf, int len, struct ip *iph);
  ```

More than one callback of each type can be registered.
TCP state information is accessed via a structure

```
struct tcp_stream *ns
```

Most important fields:

```
struct tuple4 addr:  a TCP address 4-tuple
char nids_state:   socket state
    NIDS_JUST_EST: connection has just been established
    NIDS_DATA:    new data has arrived
    NIDS_CLOSE, NIDS_RESET, NIDS_TIMEOUT: connection has been closed
struct half_stream client, server: per-direction state information (data, number of arrived bytes, etc.)
void *user: pointer to a persistent (per connection) user data
```
Advantages:
- The only known library for writing NIDS applications.
- Robust defragmentation and TCP re-assembly
- Easy access to packet/connection payload

Disadvantages:
- Moderate performance (<100 Mbit/s)
- Poor access to packet headers
- Poor documentation
- Terrible code

Alternatives: hacking into existing IDS (Snort or Bro)
Lessons learned

- The main difficulty in writing network sensors is the need to mimick (relatively complex) protocol stacks.
- Packet-level capture is relatively well understood, and an effective solution is given by libpcap.
- Connection and flow-level capture is rather messy, and no established universal solutions exist.