Foundation: Capturing and Parsing Packets

Abstract
This assignment lays the foundation for the weekly assignments in the Advanced Network Security course. In the weeks to come, you will develop key concepts of an intrusion detection system, learn principles on how to work efficiently with network data, and implement defenses for various types of attacks. In comparison to the other assignments to come, this one is more difficult in terms of working with packet formats at a bit level (which requires some practice), while the remaining ones focus more on data mining, measurements and data interpretation.
This project is due Sunday 18 Feburary, 23:59.

For troubleshooting networking problems and monitoring traffic for potential security issues, it is important to get as immediate access as possible to network data. If malicious traffic is sent towards a host that triggers a vulnerability in the network stack (for example fragmentation attacks such as the "ping of death" in chapter 5.5), such abnormalities will most likely not be passed along to the application level and the attack thus be invisible for a monitoring program. In the discussion of the physical layer, you have seen that in order to obtain an exact raw copy of network traffic, organizations can use Ethernet and optical taps to inspect packets flowing across a link. Similar mechanisms to inspect packets exist in the hosts' operating systems, and these interfaces provide the foundation for troubleshooting applications and intrusion detection systems. In the following we will discuss these interfaces, review their capabilities, and use one of them to build a packet sniffer. We will then learn for the case of DNS how to translate a protocol specification into a packet parser.

As we have seen in the introduction, networking systems are typically designed as stacks of independent components in order to allow for interoperability and to simplify system engineering. While some of the lower level components such as the interaction at the physical layer are handled by hardware, the bulk of the network stack is implemented by the host's operating system. If a user application would like to make use of the network connection, it can connect to a special interface provided by the operating system as shown in figure 1, called a socket. *Sockets* come in two flavors: stream sockets create an end-to-end connection that recovers from packet loss or packet reordering in transit and is realized through TCP, a datagram socket provides a best effort connectivity by means of UDP. To the end application, none of these internal specifics are visible though; the socket interface abstracts away all of these implementation details, requiring the user application to only provide the data that should be sent to the remote party. From the perspective of the application, a write to a remote host over a network behaves much like a write to a local file. In the Berkeley socket API which is for example provided by the C language on Linux or Unix, an application may request a socket through the function `socket()`, and subsequently call `send()` or `receive()` on the socket to transmit and receive data from the network connection.

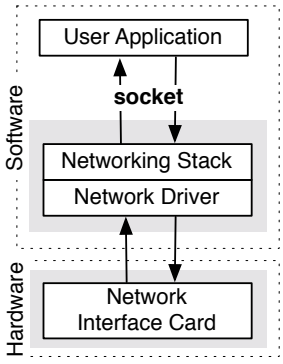

Figure 1

For network monitoring and security purposes, stream and datagram sockets are clearly unsuitable. First, they encapsulate and hide away all of the implementation details and header values of the network connection at the lower network layers, important features we would like to evaluate. Second, as we are limited to TCP/IP and UDP/IP packets via the stream and datagram interface, we can neither send nor receive other packet types with it. Thus, we could not receive any control information from the network layer such as ICMP, nor implement important tools such as ping using this interface. The Berkeley socket API provides however another socket type with more visibility on the packet, the *raw socket* by specifying the socket type as `SOCK_RAW`. The availability of raw sockets heavily varies by operating system, Windows for example provides only a marginal raw socket API, while raw sockets on Linux or Unix are standard. Instead of limiting the interaction to only the application user data, raw sockets allow access to the header information of either the TCP or UDP transport layer as shown in figure 2. If you open the raw socket with the configuration `IPPROTO_RAW`, also access to layer 3 is enabled, making it possible to set any IP header value and implement any arbitrary payload in it. As this opens up the potential for misuse – packets may for example be sent with a false source IP address –, raw sockets generally require elevated access privileges. While raw sockets allow much flexibility for sending arbitrary packets, receiving frames is somewhat complicated. A socket at the transport layer can re-

ceive incoming frames addressed to this layer, while a socket opened at the network layer will not obtain any incoming packets as they are delivered by the networking stack to the handlers in the network protocol stack.

A packet sent by a raw socket is next processed by the operating system's link layer, where the MAC header and trailer are added. As a result, a manipulation of link layer headers is not possible using a raw socket, likewise any link layer information will be stripped from an incoming frame before being delivered to the network layer and thus also not be visible to the raw socket. In many cases, it is however necessary to be able to also read and set link layer headers, for example to monitor the link level interaction on a LAN or follow low level protocols such as ARP which we will do in the next part of the term project. Some operating systems such as Linux introduce a lower level alternative in the form of *packet sockets*, a socket type with access to the link layer through which the link layer header and trailer and all the payload information may be edited. Like raw sockets, also packet sockets are not universally available across operating system platforms. Link layer access is thus commonly implemented by means of special libraries, one commonly used interface is *pcap* (*p*acket *cap*ture), a platform-independent API to obtain and inject raw network packets which is employed by a number of application you have come across in this book such as wireshark, snort, or tcpdump. In the first part of your term project, we will use the pcap API to receive and read incoming frames from the network.

Link layer access mechanism provide two very important benefits to network security monitoring. First, while some operating systems might only process standard protocols such as IP, TCP, UDP or ICMP, link layer monitoring allows the tracking of proprietary protocols, corrupted frames or packets unknown to the operating system. Second, link layer access allows the operation of a network interface in *promiscuous mode*. When packets are delivered from the link layer to the network layer, the IP stack will discard all frames not addressed to the local IP address. In result, a raw socket may only see packets addressed to the local host. By turning the network adapter into promiscuous mode, the link layer will decode any incoming network frame and pass it along to a link layer-based listener, which allows a monitoring of all local area network traffic within the capabilities of the physical layer medium.

The Berkeley Packet Filter and the pcap API

Whenever a packet arrives from the physical media and can be successfully decoded by the network interface card (NIC), the NIC determines based on the included link layer address whether the packet should be delivered to the local host. If a packet is bound for the host's link layer address and was sent to a broadcast address, an interrupt is triggered in the host's operating system. The operating system kernel then timestamps the packet, and copies it from the network card into a reserved memory location inside the OS kernel, and determines based on the type of packet and header information which subroutine will process it further. In case of Ethernet frames, the EtherType header indicates the type of protocol that is encapsulated in the payload. An EtherType value of 0x0800 instructs the host that an IPv4 packet is wrapped in the frame payload, a value of 0x86DD an IPv6 packet. Based on this information, the host the passes the packet on to the right networking stack for processing, which in turn will deliver the payload to the corresponding host application.

The operating system may deliver the packet not only to the appropriate network stack, but provide additional copies to other consumers in the kernel. Obtaining raw link layer access for each application that is diagnosing and inspecting network traffic is however somewhat impractical:

- a kernel module would be developed for each application, which needs to be maintained and adapted for different operating system and kernel versions,
- as not all applications would need access to every type of traffic, obtaining a raw feed every time will require extra logic in applications to parse, filter and discard irrelevant data, and
- for security reasons we should avoid doing any processing in the privileged kernel space, so that possible faults such as a buffer overflow will have minimal consequences.

To avoid these shortcomings, we can fall back on the Berkeley Packet Filter (BPF), which provides an universal interface for raw access to the link layer and obtains a copy of every packet that is processed by the NIC driver, both incoming and outgoing. BPF also makes it possible to send arbitrary packets at the link layer, and in case the network interface card allows this configuration can monitor the physical media in *promiscuous mode* to retrieve *all* packets on the medium, not only those addressed to the local host. As raw packet capture can be risky and may not even be desired, the Berkeley Packet Filter allows interfacing applications to specify which type of packets they are interested in and only deliver those via the API, – if your application is only interested in TCP handshakes processing all other traffic will unnecessarily slow it down.

As shown in figure 3, BPF may be accessed directly by applications operating in user space via raw sockets (which can now receive TCP/UDP packets via BPF as we have bypassed the networking stack), or via the pcap API which introduces a platform-independent high-level API for packet capture and processing. The pcap API is exported in Unix and Linux by the libpcap library, in Windows an equivalent is provided by winPcap. Although libpcap is a C and C++ API, there are wrappers for other languages such as Java, Python or C#. As it is the goal of this term project to train you in the practical and operational issues of network security, please choose the language that you are most comfortable working with. As you will see when you discuss with your classmates about

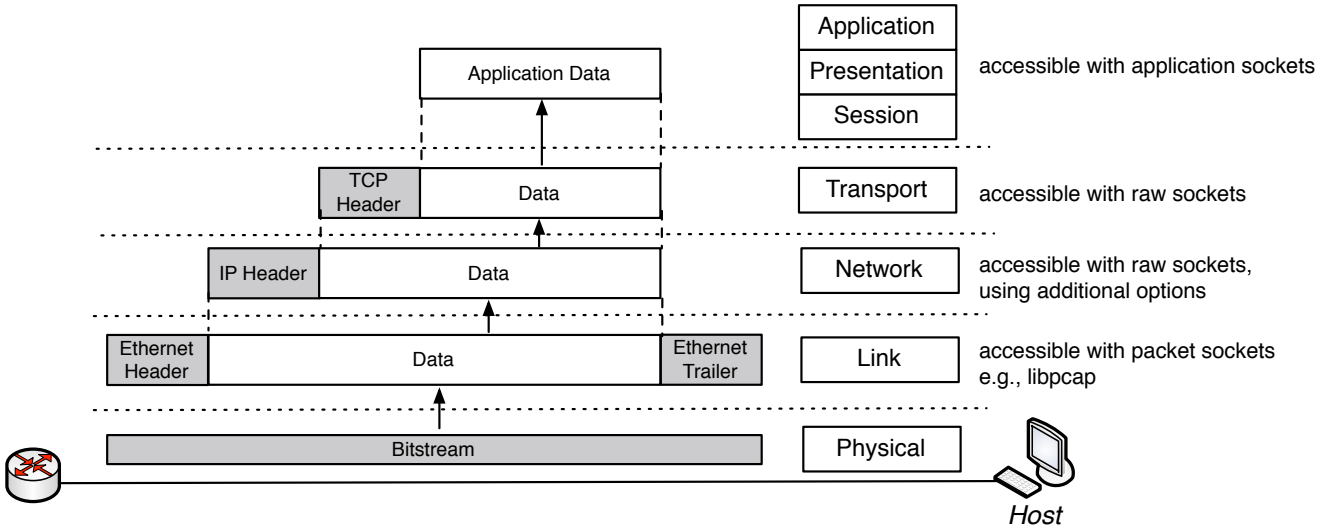

Figure 2. Socket types and data visibility.

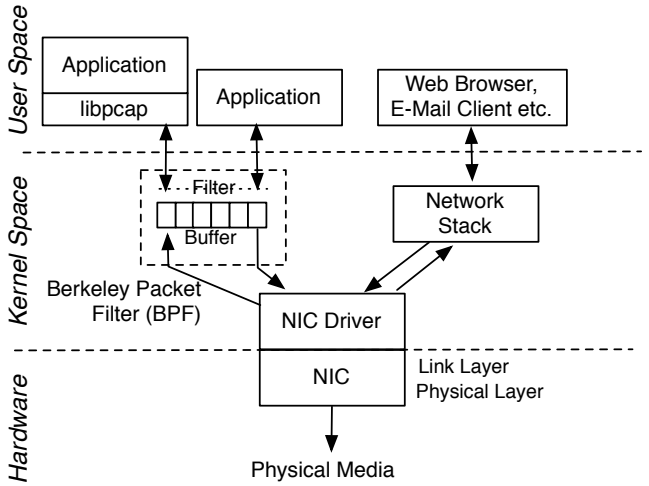

Figure 3. Network data processing in the Linux operating system

the experience they had implementing their term project in another language, every language and API wrapper has its own advantages and disadvantages in parsing network data and implementing sophisticated algorithms on top of this data.

> **Task 1** *Install in your operating system the library required for the pcap API. Research available API bindings for the language of your choice, and create a program that can capture live data from the NIC, save packets to a pcap file, and read from a pcap file.* ∎

> **Task 2** *Browse the Berkeley Packet Filter manual for the BPF syntax and familiarize yourself with its filtering capabilities. Describe two ways to filter for only TCP packets.* ∎

Parsing Packets

In the second part of the first term project assignment, you will practice to translate protocol specifications into code and write parsers for network traffic. We will use the DNS protocol in this exercise, as in a later stage of the term project you will develop a component to detect malicious DNS traffic, and as the DNS packet format includes interesting aspects that will prepare you to deal with most other protocols that you will encounter later.

Figure 4 shows the header of a DNS packet, which is discussed in chapter 5.6. The fields in the header are defined as follows:

- *QueryID.* (16 bit) Numeric identifier to match incoming responses to previous requests.
- *QR.* (1 bit) Flag to differentiate a query (0) from response (1).
- *Opcode.* (14 bit) Numeric identifier to specify the type of query. 0 means a standard query, 1 an inverse query, 2 a server status request.
- *AA.* (1 bit) Flag to indicate that the name server is an authority for the domain name in the question section. If the answer contains multiple domain names, the bit applies to the domain name listed in the question or the first record in the answer.
- *TC.* (1 bit) Flag to indicate that the reply only contains the first 512 bytes of the answer.
- *RD.* (1 bit) Flag set in a query to indicate that the question should be resolved recursively.
- *RA.* (1 bit) Flag indicates whether the name server offers recursive query lookup.
- *000.* These three bits are reserved for future use. In recent versions of the DNS protocol, this space is used by flags to indicate whether the data was authenticated. We will ignore DNSSec for now in the packet parser.

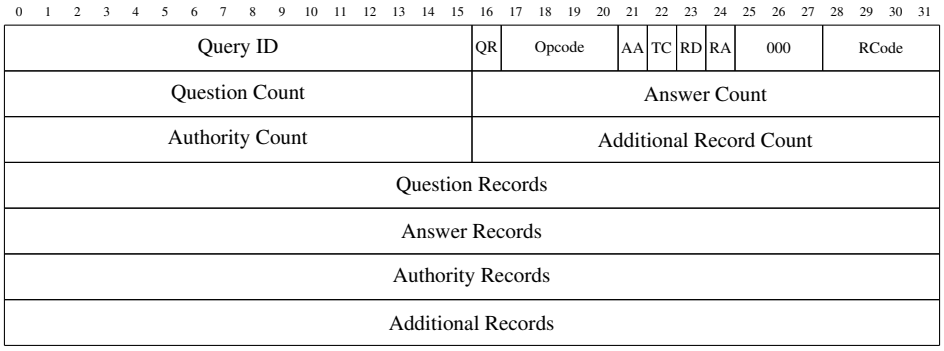

Figure 4. General format of a DNS packet

- *Rcode.* (4 bit) Field carrying a status code. A value of 0 means no error, 1 a format error, 2 a server failure, 3 a name error which indicates for an authoritative name server that the domain does not exist. A response code of 4 is returned if the name server does not support this query, 5 if the name server does not permit this query. Additional return codes are listed in RFC2136, but will be ignored as part of this assignment.

- *Questions/Answer/Authority/Additional Records Count.* (16 bit each) Unsigned value for the number of questions, answers, authoritative answers and additional record answers contained in the variable-length fields below.

When you study the DNS packet header, it becomes evident that you need to extract data and translate it from the network packet in multiple ways. Flags that are 1 bit long can be individually mapped to a boolean value, although in practice flags are usually grouped together and aligned with an 8-bit boundary so that they can be read together and simultaneously assigned to multiple values. You see this also in the DNS packet header where five flags and the opcode occupy a byte from bit 16 on. When a value is larger than 8 bits some complications arise, as data could be formated in *big endian* format with the most significant byte transferred first or in *little endian* format, where the most significant byte is the right most value. For historical reasons and to avoid confusion, network traffic are encoded as big endian, whereas the commonly used x86 architecture is *little endian*. Consider for example the 16 bit decimal value 773, or 0x0305 in hexadecimal ($3 \cdot 16^2 + 0 \cdot 16^1 + 5$), which you would find as `03 05` in a network packet but written as `05 03` in the memory of an x86 computer. This means that for values exceeding a byte a translation is necessary when reading or writing to the network at a low level. Languages such as C provide handy functions to convert between these formats depending on the architecture, the `htons` function – an abbreviation for host-to-network short – translates a 16-bit short value from the format used by the local host to the big endian network byte order, while `ntohs` translates in reverse. 32 bit longs can be mapped using `ntohl` and `htonl`. In Linux on an x86 machine, `htons` would flip the order of the two bytes, when running on a big endian architecture such as a PowerPC these functions are empty.

After the specification of the number of questions, answers, authority and additional records to be found in the packet, the remainder of the DNS packet payload then contains each of these records, always aligned to the 32 bit word boundary. The layout, formatting and interpretation of the questions and answer records is described in sections 4.1.2 and 4.1.3 of RFC1035, the specification of the DNS protocol from November 1987. Lookup the structure of these records from this RFC.

Task 3 *Extend your packet capture program with the means to parse packets belonging to the DNS protocol. To simplify the assignment, we will only consider the functionality of DNS as described in RFC1035, and thus exclude any later additions such as DNSSec.*

As you are working in an "open" environment and your program needs to be able to work with network packets that contain mistakes or contain maliciously crafted data from an adversary, you must make sure that it works correctly with incorrect or unknown data. As a first step, make sure that it correctly parses the main types of this RFC, but ignores and gracefully responds to other record types that are not part of your program. We will ignore these additions for now to simplify the assignment. If your parser receives packets that are not DNS packets, it must ignore them without failing. You can test your implementation by capturing your own DNS traffic, or by parsing the example pcap file you have been provided. ∎