**Assignment No. : \_**

**Title:**Packet sniffer

**Aim:**Implement Packet sniffer program to identify header of various protocols.

**Objective:**

1. To study and Intercept the network traffic flowing in and out of a system through network interfaces.
2. To capture relevant packet from protocol.
3. To study and analyze various protocols headers.

**Theory:**

**Packet Sniffer:** Packet sniffing is used within a network in order to capture and register data flows. Packet sniffing allows you to discern each individual packet and analyze its content based on predefined parameters.Packet sniffing allows for very detailed network monitoring and bandwidth usage analysis. It, however, requires a broader knowledge of networks and their inner functions, in order to be able to recognize the relevance of the data being monitored.

A packet sniffer (also known as a network analyzer, protocol analyzer or for particular types of networks, an Ethernet sniffer or wireless sniffer) is a computer program or a piece of computer hardware that can intercept and log traffic passing over a digital network or part of a network. As data streams flow across the network, the sniffer captures each packet and, if needed, decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content.

Packet sniffers work by intercepting and logging network traffic that they can 'see' via the wired or wireless network interface that the packet sniffing software has access to on its host computer.

**Working of Packet Sniffer:**

On a wired network, what can be captured depends on the structure of the network. A packet sniffer might be able to see traffic on an entire network or only a certain segment of it, depending on how the network switches are configured, placed, etc. On wireless networks, packet sniffers can usually only capture one channel at a time unless the host computer has multiple wireless interfaces that allow for multichannel capture.Once the raw packet data is captured, the packet sniffing software must analyze it and present it in human-readable form so that the person using the packet sniffing software can make sense of it.

The person analyzing the data can view details of the 'conversation' happening between two or more nodes on the network. Network technicians can use this information to determine where a fault lies, such as determining which device failed to respond to a network request.

Hackers can use sniffers to eavesdrop on unencrypted data in the packets to see what information is being exchanged between two parties.They can also capture information such as passwords and authentication tokens (if they are sent in the clear). Hackers can also capture packets for later playback in replay, man-in-the-middle, and packet injection attacks that some systems may be vulnerable to.

**Need of Sniffer:**

 On a normal LAN there are thousands of packets exchanged by multiple machines every minute, ample supply for any attacker.

 Anything transmitted in plaintext over the network will be vulnerable - passwords, web pages, database queries and messaging to name a few.

 A sniffer can easily be customized to capture specific traffic like telnet sessions or e-mail. Once traffic has been captured, crackers can quickly extract the information they need - logins, passwords and the text of messages.

 And the users will likely never know they were compromised - sniffers cause no damage or disturbance to a network environment.

**What is Scapy:**

Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery (it can replace hping, 85% of nmap, arpspoof, arp-sk, arping, tcpdump, ethereal, p0f, etc.). It also performs very well at a lot of other specific tasks that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining technics (VLAN hopping, +ARP cache poisoning, VOIP decoding on WEP encrypted channel, etc.). Scapy provides a [Python](https://en.wikipedia.org/wiki/Python_(programming_language)) interface into [libpcap](https://en.wikipedia.org/wiki/Pcap" \o "Pcap), (WinPCap on Windows), in a similar way to that in which [Wireshark](https://en.wikipedia.org/wiki/Wireshark" \o "Wireshark) provides a view and capture [GUI](https://en.wikipedia.org/wiki/GUI). It can interface with a number of other programs to provide visualisation including [Wireshark](https://en.wikipedia.org/wiki/Wireshark" \o "Wireshark) for decoding packets, [GnuPlot](https://en.wikipedia.org/wiki/GnuPlot" \o "GnuPlot) for providing graphs, [graphviz](https://en.wikipedia.org/wiki/Graphviz" \o "Graphviz) or [VPython](https://en.wikipedia.org/wiki/VPython" \o "VPython) for visualisation, etc.

**Concurrent and multicore programming:**

Single-threaded programs execute one line of code at a time, then move onto the next line in sequential order (except for branches, function calls etc.). This is generally the default behaviour when you write code. Multi-threaded programs are executing from two or more locations in your program at the same time (or at least, with the illusion of running at the same time). For example, suppose you want to perform a long file download from the internet, but don’t want to keep the user waiting while this happens. Imagine how inconvenient it would be if we couldn’t browse other web pages while waiting for files to download! So, we create a new thread (in the browser program for example) to do the download. In the meantime, the main original thread keeps processing mouse clicks and keyboard input so you can continue using the browser. When the file download completes, the main thread is signaled so it knows about it and can notify the user visually, and the thread performing the download closes down.

**How do threads work in practice?**

Most programs start off running in a single thread and it is up to the developer to decide when to spin up (create) and tear down (destroy) other threads. The general idea is:

Application calls the system to request the creation of a new thread, along with the thread priority (how much processing time it is allowed to consume) and the starting point in the application that the thread should execute from (this is nearly always a function which you have defined in your application).Main application thread and secondary thread (plus any other created threads) run concurrentlyWhen the main thread’s work is done, or if at any point it needs to wait for the result of a secondary thread, it waits for the relevant thread(s) to finish. This is (misleadingly) called a join operation.

Secondary threads finish their work and may optionally signal to the main thread that their work is done (either by setting a flag variable, or calling a function on the main thread)

Secondary threads close down if the main thread was waiting on a join operation (see step 3), the termination of the secondary threads will cause the join to succeed and execution of the main thread will now resume

**Threading on multi-core systems**

If your processor has multiple cores, and your operating system supports it, creating a new thread will (or may by default) cause the new thread to be executed on an unused or least busy core. Therefore, running multi-threaded applications on multi-core systems is the primary way to take advantage of multiple cores and can enable a several-fold performance increase on complex tasks. If your application is multi-threaded, you can be mostly assured that it will automatically take advantage of multi-core processors. In contrast, on single-core processors, the threads will be time-sliced by the operating system in the same way processes are in a multi-tasking environment and all run on the single core, so there is no effective performance gain.

Note that there are laws of diminishing returns at work when using something like the parallel aggregation pattern: creating 4 worker threads will not necessarily lead to a 4-fold performance increase. Depending on the algorithms in use, the amount of co-ordination between threads, overhead from accessing memory shared between threads and other factors, the speed-up will be sub-linear. For more information see Amdahl’s law. Note also that algorithms which are well-designed for sequential (single thread/core) use are not necessarily optimized for multi-threaded/multi-core use and may need to be significantly re-designed.

**Thread safety:**

Sometimes running several parts of your program at once can lead to unexpected behaviour or even cause memory corruption and crashes if you do not take precautions. If the same function can be executed in several threads simultaneously and independently without affecting each other, and additionally is designed to co-operate with other threads when accessing or changing their variables etc., the function is said to be thread safe.

Creating thread safe code is essentially the biggest problem and stumbling block for developers getting to grips with multi-threaded programming. Even experienced developers can come up across subtle and dangerous bugs which are very difficult to understand and debug. The most common issues are summarized below.

Race conditions

Imagine you have two functions running in separate threads, each writing their actions to a log (C++; we use the standard console output below but the principle is the same):

*void myThread1()*

*{*

*while (true)*

*{*

*// do some work*

*// log it:*

*std::cout<< "Hello World" <<std::endl;*

*}*

*}*

*void myThread2()*

*{*

*while (true)*

*{*

*// do some work*

*// log it:*

*std::cout<< "The quick brown fox" <<std::endl;*

*}*

The order of execution of the two threads, and when the threads get time-sliced and control passed to the other thread is unpredictable. This is acceptable and usually what you want, since the tasks are to run independently. You might expect an output like:

*Hello World*

*Hello World*

*The quick brown fox*

*Hello World*

*The quick brown fox*

*The quick brown fox*

*The quick brown fox*

*Hello World*

*Hello World*

or any other random combination of output of these two lines of text depending on how the threads are time-sliced. Unfortunately, things are not so simple. The thread may get time-sliced during the call to cout, and the other thread could output an entire line (or more) before the other thread’s call to cout is allowed to finish. This leads to weird output like this:

*Hello WorThe quick brown fox*

*The quick brown fox*

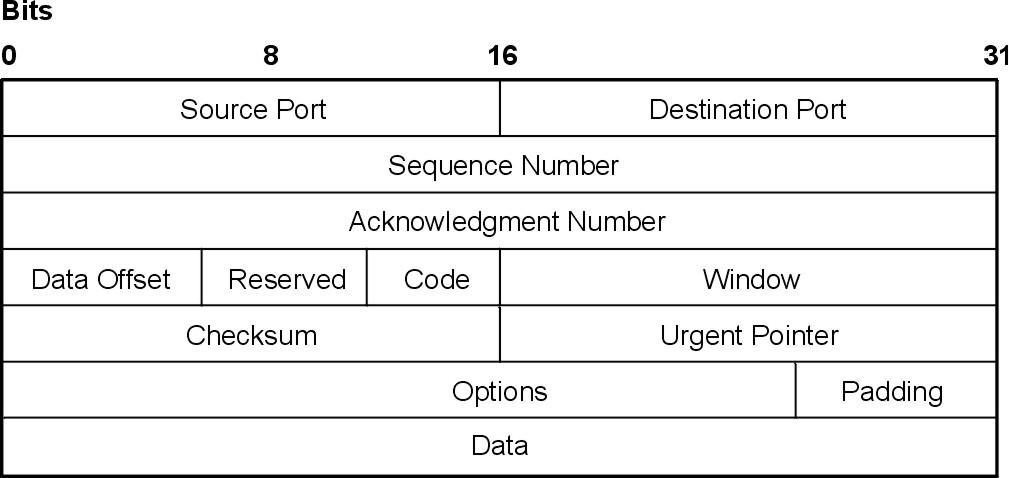
*ld*

This is almost certainly not what you want. Here the problem is quite harmless, but in real applications this can easily cause file or database corruption if two threads are writing to the same output (as they are above, writing to the console).

Race conditions are essentially when an operation which is meant to be atomic – that is to say, it should be completed as a single un-interruptible operation, not started at all, or rolled back to the previous state if completion failed – is indeed interrupted part-way through. In the example above, we incorrectly assume that writing to the console (or a log file) is an atomic operation, and we expect it to behave as an atomic operation, but in fact it does not.

A similar situation can arise with database tables, variables, or any other storage medium. Two threads updating the same rows in a database pseudo-simultaneously without protecting against a race condition may lead to the rows being partly updated with the data generated by one thread, and partly from the data by another (depending on the database architecture). Obviously this is a very dangerous result as the updated rows may be inconsistent. There are various ways to solve this problem but the most common is by use of so-called synchronization objects.

**Various Internet Protocols:**

**1) TCP:**The Transmission Control Protocol provides a communication service at an intermediate level between an application program and the Internet Protocol. It provides host-to-host connectivity at the [Transport Layer](http://en.wikipedia.org/wiki/Transport_Layer) of the Internet model. An application does not need to know the particular mechanisms for sending data via a link to another host, such as the required packet fragmentation on the transmission medium. At the transport layer, the protocol handles all handshaking and transmission details and presents an abstraction of the network connection to the application.

**Source port (16 bits):** Identifies the sending port

**Destination port (16 bits):** Identifies the receiving port

**Sequence number (32 bits)**

Has a dual role:

* If the SYN flag is set (1), then this is the initial sequence number. The sequence number of the actual first data byte and the acknowledged number in the corresponding ACK are then this sequence number plus 1.
* If the SYN flag is clear (0), then this is the accumulated sequence number of the first data byte of this segment for the current session.

**Acknowledgment number (32 bits):** if the ACK flag is set then the value of this field is the next sequence number that the receiver is expecting. This acknowledges receipt of all prior bytes (if any). The first ACKsent by each end acknowledges the other end's initial sequence number itself, but no data.

**Data offset (4 bits):** specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes, allowing for up to 40 bytes of options in the header. This field gets its name from the fact that it is also the offset from the start of the TCP segment to the actual data.

**Reserved (3 bits):** for future use and should be set to zero

**Flags (9 bits) (aka Control bits)**

Contains 9 1-bit flags

* NS (1 bit) – ECN-nonce concealment protection (experimental: see [RFC 3540](http://tools.ietf.org/html/rfc3540)).
* CWR (1 bit) – Congestion Window Reduced (CWR) flag is set by the sending host to indicate that it received a TCP segment with the ECE flag set and had responded in congestion control mechanism (added to header by [RFC 3168](http://tools.ietf.org/html/rfc3168)).
* ECE (1 bit) – ECN-Echo has a dual role, depending on the value of the SYN flag. It indicates:
* If the SYN flag is set (1), that the TCP peer is [ECN](http://en.wikipedia.org/wiki/Explicit_Congestion_Notification) capable.
* If the SYN flag is clear (0) that a packet with Congestion Experienced flag in IP header set is received during normal transmission (added to header by [RFC 3168](http://tools.ietf.org/html/rfc3168)).
* URG (1 bit) – indicates that the Urgent pointer field is significant
* ACK (1 bit) – indicates that the Acknowledgment field is significant. All packets after the initial SYN packet sent by the client should have this flag set.
* PSH (1 bit) – Push function. Asks to push the buffered data to the receiving application.
* RST (1 bit) – Reset the connection
* SYN (1 bit) – Synchronize sequence numbers. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid for when it is set, and others when it is clear.
* FIN (1 bit) – No more data from sender

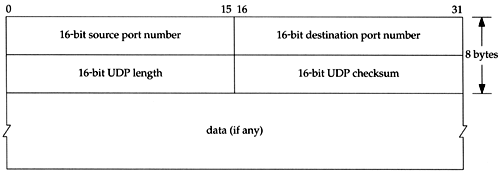
**Window size (16 bits):** The size of the *receive window*, which specifies the number of window size units (by default, bytes) (beyond the sequence number in the acknowledgment field) that the sender of this segment is currently willing to receive.

**Checksum (16 bits):** The 16-bit [checksum](http://en.wikipedia.org/wiki/Checksum) field is used for error-checking of the header and data

**Urgent pointer (16 bits):** If the URG flag is set, then this 16-bit field is an offset from the sequence number indicating the last urgent data byte

**2) UDP:**UDP uses a simple [connectionless](http://en.wikipedia.org/wiki/Connectionless) transmission model with a minimum of protocol mechanism. It has no [handshaking](http://en.wikipedia.org/wiki/Handshaking) dialogues, and thus exposes any [unreliability](http://en.wikipedia.org/wiki/Reliability_(computer_networking)) of the underlying network protocol to the user's program. There is no guarantee of delivery, ordering, or duplicate protection. UDP provides [checksums](http://en.wikipedia.org/wiki/Checksums) for data integrity, and [port numbers](http://en.wikipedia.org/wiki/Port_numbers) for addressing different functions at the source and destination of the datagram.

**Source port number:** This field identifies the sender's port when meaningful and should be assumed to be the port to reply to if needed. If not used, then it should be zero. If the source host is the client, the port number is likely to be an ephemeral port number. If the source host is the server, the port number is likely to be a well-known port number.



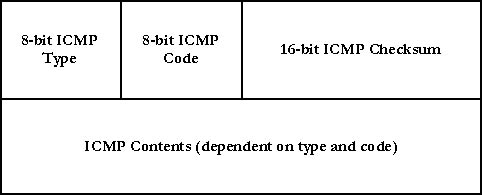
**Destination port number:** This field identifies the receiver's port and is required. Similar to source port number, if the client is the destination host then the port number will likely be an ephemeral port number and if the destination host is the server then the port number will likely be a well-known port number.

**Length:** A field that specifies the length in bytes of the UDP header and UDP data. The minimum length is 8 bytes since that's the length of the header.

**Checksum**

The [checksum](http://en.wikipedia.org/wiki/Checksum) field is used for error-checking of the header *and* data. If no checksum is generated by the transmitter, the field uses the value all-zeros.[[6]](http://en.wikipedia.org/wiki/User_Datagram_Protocol#cite_note-rfc768-6) This field is not optional for IPv6

**3) ICMP:**It is used by network devices, like routers, to send error messages indicating, for example, that a requested service is not available or that a host or router could not be reached. ICMP can also be used to relay query messages. It is assigned protocol number 1. ICMP differs from transport protocols such as [TCP](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) and [UDP](http://en.wikipedia.org/wiki/User_Datagram_Protocol) in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications.



**Type:** ICMP type

**Code:** ICMP subtype

**Checksum:** Error checking data, calculated from the ICMP header and data, with value 0 substituted for this field. The Internet Checksum is used, specified in [RFC 1071](http://tools.ietf.org/html/rfc1071).

**Rest of Header:** Four-bytes field, contents vary based on the ICMP type and code.

**Conclusion:** Thus, we have successfully studied and extracted header of different protocols using concept of Packet Sniffer.

**FAQs:**

1. What are the different uses of Packet Sniffer?
2. Draw Header Formats of different Protocols like IP, IGMP, SCTP and DNS.
3. List various existing tools available for Packet Sniffing.