Sendit

Release 1.0.5

Aug 18, 2020

Contents:

Sendit is a Python library for handcrafting, sending, and receiving packets. You can modify any value in Ethernet, ARP, IPv4, IPv6, TCP, and UDP protocols and send it. This allows you to send and receive data as a different MAC and/or IP address, do things such as mapping out a network using ARP, modify values to prevent OS fingerprinting, and so much more. While Sendit works at layers 2 to 4, meaning we are working with frames, packets, and segments, for purposes of simplicity in this documentation all units of data will be referred to as packets, their layer specified by the protocol being discussed.

CHAPTER 1

sendit

1.1 sendit package

1.1.1 Subpackages

1.1.1.1 sendit.applications package

Submodules

sendit.applications.arp_daemon module

sendit.applications.arpmap module

Sprays ARP requests across provided subnet, prints replies

sendit.applications.arpmap.**ARP_map**(*network*, *prefix*, *interface*, *mac*, *ip*, *rand=False*, *delay=0.0*)

Sends out ARP requests across a network, effectively allowing a user to map out hosts on local subnet Can be used to root out hosts with duplicate IPs

Parameters

- **interface** (*String name of interface*) – Name of interface to send ARPs out of
- **network** (*String*) – Network IP address
- **prefix** (*int*) – prefix of subnet
- **mac** (*String*) – string MAC address of intended source of ARP requests. Not neccessarily MAC of your NIC
- **ip** (*String*) – string IP address of intended source of ARP requests. Not neccessarily IP of your NIC

- **rand** (*Boolean*) – boolean of whether to send ARP requests to targets randomly or in order
- **delay** (*int*) – delay between requests in seconds. Be aware of host operating system's limitations on minimum sleep time

Raises

- **TypeError** – if sock not valid Socket object
- **ValueError** – if prefix not between 8 and 32 inclusive, or if delay is negative or of network is not valid IPv4 address or if ip is not valid IPv4 address or if mac is not valid MAC address

sendit.applications.arpmap.**ARP_recv**(*nic*)
Function that receives and prints out ARP replies: MAC address, IP address, and Manufacturer of MAC

Parameters **nic** (Raw_NIC) – nic to receive ARP replies on

sendit.applications.arpmap.**print_banner**()
prints pretty banner for ARP MAP

sendit.applications.basic_examples module

Provides example usage of the sendit module

sendit.applications.ipv4_handler module

sendit.applications.ipv6_handler module

Module contents

1.1.1.2 sendit.handlers package

Submodules

sendit.handlers.arp_handler module

Creates class that listens and responds to ARP messages

class sendit.handlers.arp_handler.**ARP_Handler**(*reply=True*, *mappings=None*, *send_down=None*, *recv_up=None*)
Bases: sendit.handlers.handler.Handler

Creates class that listens and responds to ARP messages. Child class of Handler

Parameters

- **reply** (*Boolean*) – boolean of whether to answer ARP requests, defaults to True
- **mappings** (*dictionary with String keys and values, defaults to None*) – dictionary mapping MAC addresses to IPv4 addressses defaults to None. Required if reply is True
- **send_queue** (*asyncio.Queue*) – asyncio.Queue that will be used to put frames in to send

Raises **ValueError** – when reply is set to true but mappings is not defined or when keys in mappings are not valid IPv4 addresses or when values in mappings are not valid MAC addresses

listen()
listens and responds for ARP messages coming in on recv_queue, put there by an ethernet_handler

sendit.handlers.ethernet_handler module

Creates class that listens and responds to Layer2 Etherframes

class sendit.handlers.ethernet_handler.**EtherFrame_Handler**(*send_up=None*, *send_down=None*, *recv_up=None*, *recv_down=None*)

Bases: sendit.handlers.handler.Handler

Asynchronously listens for Etherframes in own queue placed there by Bytes_Handler Creates EtherFrames from raw bytes, places those in async queues based on MAC mappings in queue_mappings

Parameters

- **send_up** (*Dictionary where keys are strings in format mac_protocol, values are lists of asyncio.queue, Dictionary where keys are strings in format mac_protocol, values are lists of asyncio.queue*) – asyncio.Queue OR dictionary of queues to put items in to go to higher layers, dictionary mapping async queues to protocol names
- **send_down** (*asyncio.Queue*) – asyncio.Queue to put items in to go to lower layers
- **recv_up** (*asyncio.Queue*) – asyncio.Queue to receive items from higher layers
- **recv_down** (*asyncio.Queue*) – asyncio.Queue to receive items from lower layers

add_queue(*mac*, *queue*)
Add a higher protocol layer queue into ethernet handler for management

Parameters **mac** (*String*) – mac address to listen on

await_from_higher()
Wait for frames from higher layers that needs Ethernet Header adjusted Swaps src and destination

listen()
Asynchronously listen for etherframes put in self.recv_queue If the protocols in the frame match one of the keys in self.queue_mappings Pass that frame on to all queues in corresponding list

remove_protocol(*protocol*)
Remove protocol from list of protocols that handler will handler for

Parameters **protocol** (*String*) – string of protocol to remove

remove_queue(*mac*, *protocol*)
Remove upper layer queue from Ethernet_Handler

Parameters

- **mac** (*String*) – string of mac address to remove corresponding handler
- **protocol** (*String*) – protocol of handler to remove

Raises **ValueError** – if mac not valid MAC address

sendit.handlers.ipv4_handler module

Creates class that listens and responds to Layer 3 IPv4

class sendit.handlers.ipv4_handler.**IPv4_Handler**(*send_up=None*, *send_down=None*, *recv_up=None*, *recv_down=None*)

Bases: sendit.handlers.handler.Handler

IPv4 Handler that is child class of Handler

Parameters **send_up** (*asyncio.Queue or dictionary of asyncio.queues*) – asyncio.Queue OR dictionary of queues to put items in to go to higher layers

:param send_down : asyncio.Queue to put items in to go to lower layers :type send_down: asyncio.Queue :param recv_up: asyncio.Queue to receive items from higher layers :type recv_up: asyncio.Queue :param recv_down: asyncio.Queue to receive items from lower layers :type recv_down: asyncio.Queue

await_from_higher()

Wait for frames from higher layers that needs IPv4 header adjusted Swaps src and destination and resets length and checksum

ip_fragmentation_handler(*frame*)

This pieces back together fragmented packets. This is a modified version of the algorithm defined in RFC 815 https://tools.ietf.org/html/rfc815

Parameters **frame** (EtherFrame) – ethernet frame that contains fragmented packet

Returns returns None if packet not completely defragmented, or IPv4 of defragged packet

:rtype:: IPv4 or None

listen()

Listens for frames coming in from queue, placed there by a Layer2 Handler If incoming frame contains IPv4 address destination contained in self.ips they are then passed to their respective higher level listeners Otherwise, they are discarded If frames come in with IPv4 fragmented, they are sent to ip_fragmentation_handler to be handled

sendit.handlers.ipv6_handler module

Creates class that listens and responds to Layer 3 IPv6

class sendit.handlers.ipv6_handler.**IPv6_Handler**(*send_up=None*, *send_down=None*, *recv_up=None*, *recv_down=None*)

Bases: sendit.handlers.handler.Handler

IPv6 Handler that is child class of Handler

Parameters **send_up** (*asyncio.Queue or dictionary of asyncio.queues*) – asyncio.Queue OR dictionary of queues to put items in to go to higher layers

:param send_down : asyncio.Queue to put items in to go to lower layers :type send_down: asyncio.Queue :param recv_up: asyncio.Queue to receive items from higher layers :type recv_up: asyncio.Queue :param recv_down: asyncio.Queue to receive items from lower layers :type recv_down: asyncio.Queue

await_from_higher()

Wait for frames from higher layers that needs IPv6 header adjusted Swaps src and destination and resets length and checksum

listen()

Listen for frames coming in on queue to parse the IPv6 objects inside Asynchronous

sendit.handlers.raw_nic module

Set of classes that creates abstraction for dealing with raw_sockets

class sendit.handlers.raw_nic.**Async_Raw_NIC**(*interface*, *send_up=None*, *recv_up=None*)
Bases: sendit.handlers.handler.Handler

Child Class of Handler Creates Asynchronous Raw Socket, binds to provided interface Implements send method that works with rest of library

Parameters

- **interface** (*String*) – string name of network interface ex: eth0, wlan0. Not sure? Call ifconfig and look at interface names
- **queue** (*asyncio.Queue*) – asyncio queue to send raw bytes too
- **queue** – asyncio queue to receive outgoing bytes from

a_recv(*n_bytes*)
Asynchronously receive bytes

Parameters **n_bytes** (*int*) – Number of bytes to receive

send(*frame*, *n_bytes*)
Overrides Socket send method Attempts to use as_bytes() method that is provided by all protocol classes in this library If not a class in this libary, calls str.encode on provided frame Them sends on raw socket

Parameters **frame** (*L2 object that has as_bytes function, such as Etherframe*) – frame to send on Raw_NIC

sendall_from_queue()

class sendit.handlers.raw_nic.**Raw_NIC**(*interface*)
Bases: socket.socket

Child Class of Socket Creates Raw Socket, binds to provided interface Implements send method that works with rest of library

Parameters **interface** (*String*) – string name of network interface ex: eth0, wlan0. Not sure? Call ifconfig and look at interface names

send(*frame*)
Overrides Socket send method Attempts to use as_bytes() method that is provided by all protocol classes in this library If not a class in this libary, calls str.encode on provided frame Them sends on raw socket

Parameters **frame** (*L2 object that has as_bytes function, such as Etherframe*) – frame to send on Raw_NIC

sendit.handlers.udp_handler module

Creates class that listens and responds to Layer 4 UDP

class sendit.handlers.udp_handler.**UDP_Handler**(*ports*, *send_up=None*, *send_down=None*, *recv_up=None*, *recv_down=None*)
Bases: sendit.handlers.handler.Handler

Parameters

- **ports** (*list of ints*) –
  - list of ports to listen on

- **send_up** (*asyncio.Queue or dictionary of asyncio.queues*) – asyncio.Queue OR dictionary of queues to put items in to go to higher layers

:param send_down : asyncio.Queue to put items in to go to lower layers :type send_down: asyncio.Queue :param recv_up: asyncio.Queue to receive items from higher layers :type recv_up: asyncio.Queue :param recv_down: asyncio.Queue to receive items from lower layers :type recv_down: asyncio.Queue

await_from_higher()
Wait for frames from higher layers that needs UDP header adjusted Swaps src and destination ports and ips (for checksum) and resets length and checksum

listen()
Listen for frames coming in on queue to parse the UDP objects inside

Parameters **queue** (*Queue object*) – Queue to listen in on

Module contents

1.1.1.3 sendit.helper_functions package

Submodules

sendit.helper_functions.helper module

sendit.helper_functions.helper.**MAC_to_manufacturer**(*address*)
Takes a MAC Address and searches through CSV files from IEEE to find manufacturer they belong to

Parameters **address** (*String*) – MAC Address, bytes separated with colon (:)

Returns String of manufacturer of device with provided MAC Address Unknown if not found

Return type String

sendit.helper_functions.helper.**addr_to_bytes**(*address*)
Takes address represented in string consisting of hex characters, MAC or IPv6, and converts to bytes

Parameters **address** (*String*) – addrss to convert

Returns bytes form of address

Return type bytes

sendit.helper_functions.helper.**bytes_to_MAC**(*address*)
Converts bytes form of MAC address to String form of MAC address

Parameters **address** (*bytes*) – Bytes form of MAC address

Returns String form of MAC Address

Return type String

sendit.helper_functions.helper.**checksum**(*message*)
Calculates 16 bit checksum by 1's compliment addition between all 16 bit words in message, and then taking the 1's compliment of the sum Formula same as defined for IPv4, TCP, and UDP

Parameters **message** (*bytes*) – takes header to create checksum

Returns 16 bit checksum

Return type int

sendit.helper_functions.helper.**create_ip_int_to_protocols**()

sendit.helper_functions.helper.**create_ip_protocols_to_int**()

sendit.helper_functions.helper.**form_pseudo_header**(*src_ip*, *dst_ip*, *length*, *protocol*, *version=4*)

Form TCP/UDP pseudoheader for checksum calculation

Parameters

- **src_ip** (*String*) – source ip
- **dst_ip** (*String*) – destination ip
- **length** (*int*) – length of tcp segment, header included
- **protocol** (*String*) – L4 Protocol - currently only support tcp and udp
- **version** (*int*) – IP version, defaults to 4

Returns pseudoheader in bytes

Return type bytes

sendit.helper_functions.helper.**get_IP**(*interface*)

Finds IP of a network interface on the host Uses UNIX tools ifconfig, grep, and awk Not supported on all Operating Systems or all kernels

Parameters **interface** (*String*) – string of the interface to look for

Returns string representing MAC address of interface if OS does not support commands or interface not found, program exits with code 1

Return type String

sendit.helper_functions.helper.**get_IPv6**(*interface*)

Finds IPv6 of a network interface on the host Uses UNIX tools ifconfig, grep, and awk Not supported on all Operating Systems

Parameters **interface** (*String*) – string of the interface to look for

Returns string representing MAC address of interface if OS does not support commands or interface not found, program exits with code 1

Return type String

sendit.helper_functions.helper.**get_MAC**(*interface*)

Finds MAC address of a network interface on the host Uses Unix tools ifconfig and grep Not supported on all Operating Systems, or all kernels

Parameters **interface** (*String*) – string of the interface to look for

Returns string representing MAC address of interface if OS does not support commands or interface not found, program exits with code 1

Return type String

sendit.helper_functions.helper.**int_to_ip**(*number*)

Converts int to string IPv4 address

Parameters **number** (*int*) – int to convert to string IPv4 address

Returns String of IPv4 address

Return type String

sendit.helper_functions.helper.**ip_to_int**(*address*)

Converts string IP addrss to an int

Parameters **address** (*String*) – string IP address

Returns int representing IP address

Return type int

Raises **ValueError** – if address is not valid IPv4 addrss

sendit.helper_functions.helper.**is_hex**(*string*)
Determines if string consists solely of ascii characters that are hexidecimal characters

Parameters **string** (*String*) – string to check if hex

Returns boolean representing if the string consists of only hex ascii characters

Return type Boolean

sendit.helper_functions.helper.**is_valid_MAC**(*address*)
Determines if address if valid MAC address Checks that there are 12 characters, and all are Hex values

Parameters **address** (*String*) – value to check if valid MAC address

Returns boolean representing if address is a valid MAC address

Return type Boolean

sendit.helper_functions.helper.**is_valid_ipv4**(*address*)
Determines if address if valid IPv4 address Checks that there are 4 octets, and values are between 0 and 255, inclusive

Parameters **address** (*String*) – value to check if valid IPv4 address

Returns boolean representing if address is a valid IPv4 address

Return type Boolean

sendit.helper_functions.helper.**manufacturer_to_MAC**(*manufacturer*)
Provides a list of MAC prefixes based off provided manufacturer

Parameters **manufacturer** (*String*) – Name of manufacturer

Returns list of strings of 3 byte MAC prefixes registered to that manufacturer

Return type list of Strings

Module contents

1.1.1.4 sendit.protocols package

Submodules

sendit.protocols.arp module

Creates ARP object and provides methods to parse bytes to ARP create bytes to ARP object

class sendit.protocols.arp.**ARP**(*sha*, *spa*, *tha*, *tpa*, *hrd=1*, *pro=2048*, *hln=6*, *pln=4*, *op=1*)
Bases: object

Holds all data for ARP

Parameters

- **sha** (*String, formatted as "XX:XX:XX:XX:XX:XX"*) – Source MAC address
- **spa** (*String, formated as "XXX.XXX.XXX.XXX"*) – string of source IP address

- **tha** (*String, formatted as "XX:XX:XX:XX:XX:XX"*) – string of target MAC address
- **tpa** (*String, formated as "XXX.XXX.XXX.XXX"*) – string of target IP address"
- **hrd** (*int*) – hardware code, defaults to 1 for ethernet
- **pro** (*int*) – type of protocol address - corresponds to Ethertype values, defaults to 2048 for IPv4
- **hln** (*int*) – length of hardware address in bytes, defaults to 6 for MAC length
- **pln** (*int*) – length of protocol address in bytes, defaults to 4 for IPv4 length
- **op** (*int*) – opcode of arp message, defaults to 1 for request

Raises **ValueError** – if opcode, hrd, or pln is not between 0 and 65535 inclusive or hln or pln is not between 0 and 255 inclusive

classmethod arp_parser(*data*)

Class Method that parses group of bytes to create ARP object

Parameters **data** (*bytes*) – ARP message to parse passed in as bytes

Returns ARP instance that contains the values that was in data

Return type ARP object

as_bytes()

Converts ARP to proper format of payload bytes

Returns bytes representation of ARP message

Return type bytes

sendit.protocols.etherframe module

Creates Etherframe object and provides methods to parse bytes to Etherframe create bytes to EtherFrame object

class sendit.protocols.etherframe.**EtherFrame**(*dst*, *src*, *payload*, *ethertype='ipv4'*)

Bases: object

Holds all data of Etherframe

Parameters

- **dst** (*String*) – string of destination MAC address ex: "AB:CD:EF:01:23:45"
- **src** (*String*) – string of source MAC Address ex: "AB:CD:EF:01:23:45"
- **payload** (ARP, IPv4, IPv6, *or any or any object str.encode(payload) can be called*) – data to put into Etherframe
- **ethertype** (*String*) – String representing ethertype - defaults to "ipv4". Can be ipv4, ipv6, arp, or rarp, or a custom value consisting of 4 hex string ascii chars, such as "8035"

Raises **ValueError** – if dst not valid MAC address, if src not valid MAC address, or ethertype not supported builtin AND is not 2 bytes of a string of hex characters

as_bytes()

Converts EtherFrame to proper format of payload bytes to send on Raw_NIC If self.payload is IPv4 or ARP object, their as_bytes function is called, providing the conversion of payload to properly formated bytes to be inserted into frame to be sent on Raw_NIC If self.payload is not IPv4 or ARP object, self.payload is conver

Returns bytes representation of EtherFrame

Return type bytes

bytes_to_ethertype = {b'\x08\x00': 'ipv4', b'\x08\x06': 'arp', b'\x805': 'rarp', b'

classmethod etherframe_parser(*data*, *recursive=True*)
Class Method that parses group of bytes to create EtherFrame Object

Parameters

- **data** (*EtherFrame*) – etherframe passed in as bytes If IPv4 is type of frame, payload will be IPv4 object created If ARP is type of frame, payload will be ARP object created
- **recursive** – boolean of whether to parse recursively to higher layers, defaults to True If protocol is "IPv4", payload will be IPv4 object created If protocol is "IPv6", payload will be IPv6 object created If protocol is "ARP", payload will be ARP object created :type recursive: Boolean

Returns EtherFrame instance that contains the values that was in data

Return type *EtherFrame*

ethertype_to_bytes = {'arp': b'\x08\x06', 'ipv4': b'\x08\x00', 'ipv6': b'\x86\xdd',

parse_further_layers(*recursive=True*)
Method that parses higher layer information contained in payload

Parameters **recursive** (*boolean*) – boolean value of whether parsing function should be called recursively through all layers

Returns Object representation of payload if possible to parse, if not returns self.payload

Return type *ARP*, *IPv4*, *IPv6*, or bytes

sendit.protocols.ipv4 module

Creates IPv4 object and provides methods to parse bytes to IPv4 create bytes to IPv4 object

class sendit.protocols.ipv4.**IPv4**(*src*, *dst*, *payload*, *id=0*, *length=0*, *df=False*, *mf=False*, *offset=0*, *ttl=64*, *protocol='tcp'*, *dscp=0*, *ecn=0*, *version=4*, *checksum=0*)
Bases: object

Creates IPv4 object from parameters

Parameters

- **src** (*String*) – source IP address
- **dst** (*String*) – destination IP address
- **payload** (*TCP or UDP objects of String*) – payload of the packet
- **id** (*int*) – identification number of packet, defaults to 0
- **length** (*int*) – total length of IP packet in bytes - header + data together, defaults to 0, calculated when as_bytes called. If IPv4 object created from parser function, takes value of captured IPv4 packet, and NOT calculated in as_bytes unless reset to 0 manually with reset_calculated_fields
- **df** (*Boolean*) – do not fragment flag, default to False
- **mf** (*Boolean*) – more fragments flag - deafult to False

- **offset** (*int*) – frag offset of packet, default to 0
- **ttl** (*int*) – time to live, default to 64
- **protocol** (*String or int*) – string name of protocol carried in packet.currently supported values: "tcp", "udp", "icmp", custom int value accepted IF valid
- **dscp** (*int*) – differentiated services value - default of 0
- **ecn** (*int*) – explicit congestion notification - default of 0
- **version** (*int*) – version of IP
- **checksum** (*int*) – checksum of packet. By default, not calculated and set to 0 and to be calculated when as_bytes called. Set when IPv4 object created from parser function, and unless reset manually or with reset_calculated_fields function, will NOT be recalculated when as_bytes is called

as_bytes()
Converts IPv4 to proper format of payload bytes to send set as EtherFrame payload If self.payload is TCP or UDP object, their as_bytes function is called, providing the conversion of payload to properly formated bytes to be inserted into packet If self.payload is not TCP or UDP object, self.payload is converted to bytes with str.encode(self.payload) if possible. Otherwise, it is assumed payload is already bytes

Returns bytes representation of IPv4 Packet

Return type Bytes

classmethod ipv4_parser(*data*, *recursive=True*)
Class Method that parses group of bytes to create IPv4 Object

Parameters **recursive** – boolean of whether to parse recursively to higher layers, defaults to True If protocol is "TCP", payload will be TCP object created If protocol is "UDP", payload will be UDP object created :type recursive: Boolean

Returns IPv4 instance that contains the values that was in data

Return type IPv4 object

parse_further_layers(*recursive=True*)
Method that parses higher layers

Parameters **recursive** (*Boolean*) – Whether parsing function should be called recursively through all layers, defaults to True

reset_calculated_fields()
Resets calculated fields for IPv4 - resets length and checksum

sendit.protocols.ipv6 module

Creates IPv6 object and provides methods to parse bytes to IPv6 and create bytes to IPv6 object

class sendit.protocols.ipv6.**IPv6**(*src*, *dst*, *payload*, *next='tcp'*, *limit=64*, *flow_label=0*, *ds=0*, *ecn=0*, *version=6*, *length=0*)
Bases: object

Creates IPv6 object from parameters

Parameters

- **src** (*String*) – source IPv6 address
- **dst** (*String*) – destination IPv6 address

- **payload** (*TCP or UDP object, or String*) – payload to be encapsulated inside IPv6 packet
- **next** (*String*) – next header protocol, defaults to "tcp". "udp" and "icmp" also supported
- **limit** (*int*) – hop count limit - 0 to 255 inclusive
- **flow_label** (*int*) – label for which flow packet belongs to, defaults to 0 - which is not flow
- **ds** (*int*) – Differentiated Services field, defaults to 0
- **ecn** (*int*) – Explicit Congestion Notification value, defaults to 0
- **version** (*int*) – IP version, defaults to 6
- **length** (*int*) – length of IPv6 packet, defaults to 0 and calculated in as_bytes function. If IPv6 object created with parser method, will take value of IPv6 packet captured, and will NOT be calculated in as_bytes unless reset manually to 0 or with reset_calculated_fields function

as_bytes()
Converts IPv6 to proper format of payload bytes to send set as EtherFrame payload If self.payload is TCP or UDP object, their as_bytes function is called, providing the conversion of payload to properly formated bytes to be inserted into packet If self.payload is not TCP or UDP object, self.payload is converted to bytes with str.encode(self.payload)

Returns bytes representation of IPv6 Packet

Return type bytes

classmethod ipv6_parser(*data*, *recursive=True*)
Class Method that parses group of bytes to create IPv6 Object

Parameters

- **data** (*bytes*) – IPv6 packet passed in as bytes
- **recursive** (*Boolean*) – Boolean of whether to recursively parse; if true and if protocol is "TCP", payload will be TCP object created if protocol is "UDP", payload will be UDP object created

Returns IPv6 instance that contains the values that was in data

Return type *IPv6*

parse_further_layers(*recursive=True*)
Method that parses higher layers

Parameters **recursive** – boolean value of whether parsing funciton should be called recursively through all layers

reset_calculated_fields()
Resets all calulated fields for IPv6 - resets length

sendit.protocols.tcp module

Creates TCP object and provides methods to parse bytes to TCP create bytes to TCP object

class sendit.protocols.tcp.**TCP**(*src_prt*, *dst_prt*, *src_ip*, *dst_ip*, *window*, *payload*, *sqn=0*, *ack_num=0*, *ns=False*, *cwr=False*, *ece=False*, *urg=False*, *ack=False*, *psh=False*, *rst=False*, *syn=False*, *fin=False*, *urg_pnt=0*, *version=4*, *mss=None*, *scaling=None*, *sack_permitted=None*, *stamp=None*, *sack=None*, *offset=5*, *checksum=0*)

Bases: `object`

Forms TCP Object from parameters

Parameters

- **src_prt** (*int*) – source TCP port
- **dst_prt** (*int*) – destination TCP port
- **src_ip** (*String*) – source IP address - used for creating pseudoheader to calculate checksum
- **dst_ip** (*String*) – destination IP address - used for creating pseudoheader to calculate checksum
- **window** (*int*) – window size
- **payload** (*String*) – payload of TCP Segment
- **sqn** (*int*) – sequence Number
- **ack_num** (*int*) – Acknowledgement Number
- **offset** (*int*) – 4 byte word offset of where data starts, defaults to 5
- **ns** (*boolean*) – ns flag, defaults to False
- **cwr** (*boolean*) – cwr flag, defaults to False
- **ece** (*boolean*) – ece flag, defaults to False
- **urg** (*boolean*) – urg flag, defaults to False
- **ack** (*boolean*) – ack flag, defaults to False
- **psh** (*boolean*) – psh flag, defaults to False
- **rst** (*boolean*) – rst flag, defaults to False
- **syn** (*boolean*) – syn flag, defaults to False
- **fin** (*boolean*) – fin flag, defaults to False
- **urg_pnt** (*int*) – offset of where urgent data stops
- **mss** (*int*) – maximum segment size TCP option
- **scaling** (*int*) – window scaling factor TCP option
- **sack_permitted** (*boolean*) – boolean value of whether selective acknowledgments allowed - TCP option
- **sack** (*tuple of ints*) – tuple containing byte numbers, in order, to be passes as selective acknowledgments TCP option
- **stamp** (*tuple of ints*) – tuple containing timestamp value and time stamp error
- **checksum** (*int*) – default set to 0 and calculated when as_bytes called if 0 If TCP object created from parser function, set to checksum of captured segment and NOT recalculated in as_bytes unless set to 0 manually or by calling reset_calculated_fields function

:raise ValueError when src_prt or dst_prt not between 0 and 6553 inclusive or when sqn not between 0 and 4294967295 inclusive or when ack_number not between 0 and 4294967295 inclusive or when window not between 0 and 4294967295 inclusive or when urg_pnt not between 0 and 4294967295 inclusive or when length of sack greater than 8 or when sack contains odd number of values

as_bytes()
Converts TCP to proper format of payload bytes to send self.payload is converted to bytes with str.encode(self.payload)

Returns bytes representation of TCP

Return type bytes

create_tcp_options()
Set TCP Header options mss, sack_permitted, and scaling only used during syn and ack of handshake sack (selective acknowledgment) value cannot be set during handhake Header option combinations are: mss, sack_permitted, scaling, and stamp (timestamp) during handshake - any combination of these 4 timestamp and selective acknowledgement value during regular transmissions - any combination of these 2 Depending on what combination of these are set depends on what order they are arranged in, along with nop (No-op) bytes to fit in 32 bit word

TODO handle sack value

Returns tuple consisting of bytes of options for this TCP segment and increase to options header

Return type tuple of bytes

parse_further_layers(*recursive=True*)
Method that parses higher layers and sets the payload of calling TCP object

Parameters **recursive** (*true*) – boolean value of whether parsing function should be called recursively through all layers

classmethod parse_options(*option_bytes*)
Parses TCP header options from a series of byte

Parameters **option_bytes** (*bytes*) – series of bytes containing TCP Header options

Returns list of options to return containing [MSS, Window Scale, sack_permitted, sack_values, timestamp]

Return type list

reset_calculated_fields()
Resets calculated fields for TCP - resets checksum and length

classmethod tcp_parser(*data*, *recursive=True*)
Class method that creates TCP object

Parameters **data** – TCP segment passed in as bytes

Returns TCP object created from values in data

Return type *TCP*

sendit.protocols.udp module

Creates UDP object and provides methods to parse bytes to UDP create bytes to UDP object

class sendit.protocols.udp.**UDP**(*src_prt*, *dst_prt*, *src_ip*, *dst_ip*, *payload*, *version=4*, *length=0*, *checksum=0*)
Bases: object

Creates UDP object from parameters UDP checksum is optional and therefore not currently supported

Parameters

- **src_prt** (*int*) – source port
- **dst_prt** (*int*) – destination port
- **src_ip** (*String*) – source IP address - used for creating pseudoheader to calculate checksum
- **dst_ip** (*String*) – destination IP address - used for creating pseudoheader to calculate checksum
- **version** – version of IP being carried in - used for calculating checksum
- **length** (*int*) – length of segment, defaults to 0, calculated when as_bytes called if 0. If UDP object created from parser function, set to length of captured segment and NOT recalculated in as_bytes unless set to 0 manually or by calling reset_calculated_fields function
- **checksum** (*int*) – default set to 0 and calculated when as_bytes called if 0 If UDP object created from parser function, set to checksum of captured segment and NOT recalculated in as_bytes unless set to 0 manually or by calling reset_calculated_fields function
- **payload** (*bytes*) – payload to be carried UDP

Version type int

Raises **ValueError** – if src_prt or dst_prt is between 0 and 65535 inclusive

as_bytes()
Converts UDP to proper format of payload bytes to send self.payload is converted to bytes with str.encode(self.payload)

Returns bytes representation of UDP

Return type bytes

parse_further_layers(*recursive=True*)
Method that parses higher layers

Parameters **recursive** (*boolean*) – boolean value of whether parsing function should be called through higher layers, defaults to True

reset_calculated_fields()
Resets calcualted fields for UDP - resets checksum and length

classmethod udp_parser(*data*, *recursive=True*)
Class method that creates UDP object

Parameters **data** (*bytes*) – UDP segment passed in as bytes

Returns UDP object created from values in data

Return type *UDP*

Module contents

1.1.2 Module contents

CHAPTER 2

Project Info

- Github: https://github.com/mbaker-97/sendit
- PyPi: https://pypi.org/project/sendit/

CHAPTER 3

Installing

Install with pip

```
pip install sendit
```

CHAPTER 4

Basics

Every protocol layer is its own object. To create a datagram, we start by creating the highest layer object we are working with, then creating the next highest, and passing the first object to the second, and so on. For example:

```
from sendit.protocols.EtherFrame import EtherFrame
from sendit.protocols.IPv4 import IPv4
from sendit.protocols.TCP import TCP

payload = "The quick brown fox jumps over the lazy dog"  # String payload
l4_tcp = TCP(50000, 50001, "127.0.0.1", "127.0.0.1", 1024, payload)  # Change 1st ip␣
↪to yours, 2nd to target.
# Creates IPv4 packet:
l3 = IPv4("127.0.0.1", "127.0.0.1", l4_tcp, protocol="tcp")  # Change 1st ip to yours,
↪ 2nd to target
# Creates Etherframe:
l2 = EtherFrame("AA:BB:CC:DD:EE:FF", "00:11:22:33:44:55", l3)  # Change 1st mac to␣
↪yours, 2nd to target
```

In the above example, l2, the EtherFrame, contains l4_tcp, a TCP object, inside l3, an IPv4 object.

CHAPTER 5

Sending Data

Now that you know how to create the data, how do you send it? Sendit has a class called Raw_NIC. Raw_NIC is a wrapper class around a raw socket. All protocols have a as_bytes() function, which turns the data contained in the objects into their properly formatted bytes ready to send on the line. Calling a lower protocol's as_bytes function calls all higher protocols as_bytes functions. To take the above example, and to expand it. Using a Raw_NIC's send function automatically calls the as_bytes function of the object passed into it,

```
from sendit.protocols.EtherFrame import EtherFrame
from sendit.protocols.IPv4 import IPv4
from sendit.protocols.TCP import TCP
from sendit.handlers.raw_nic import Raw_NIC

payload = "The quick brown fox jumps over the lazy dog"  # String payload
l4_tcp = TCP(50000, 50001, "127.0.0.1", "127.0.0.1", 1024, payload)  # Change 1st ip␣
↪to yours, 2nd to target.
# Creates IPv4 packet:
l3 = IPv4("127.0.0.1", "127.0.0.1", l4_tcp, protocol="tcp")  # Change 1st ip to yours,
↪ 2nd to target
# Creates Etherframe:
l2 = EtherFrame("AA:BB:CC:DD:EE:FF", "00:11:22:33:44:55", l3)  # Change 1st mac to␣
↪yours, 2nd to target
nic = Raw_NIC("lo") # Creates raw_nic on loopback interface
nic.send(l2)
```

CHAPTER 6

Advanced Usage

For advanced usage, please read through the documentation of the modules to get a full idea of what each class offers

- genindex
- modindex

Python Module Index

S

Index

L

M

P

R

S

T

U