DDoS

1. (1) time of the first packet: 24.877402

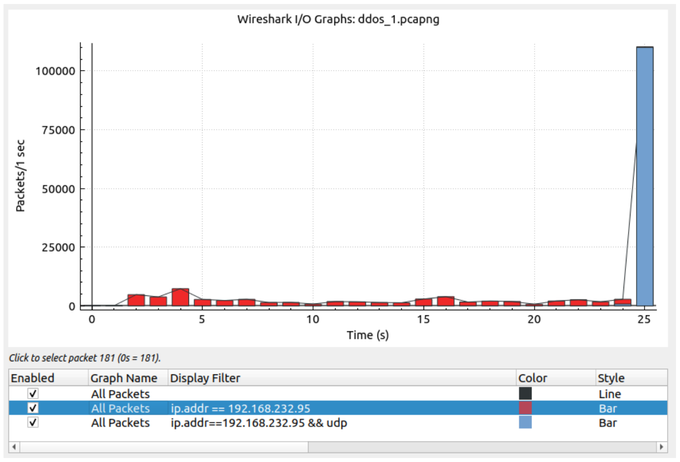
   (2) victim's IP: 192.168.232.95

2. (1) protocol: UDP

   According to Figure 1, since the behavior of UDP to the victim's IP we observe in I/O graph is almost identical to the huge burst of all the traffic, we can infer that attackers exploit UDP in this attack.

   (2) size: 482 bytes.

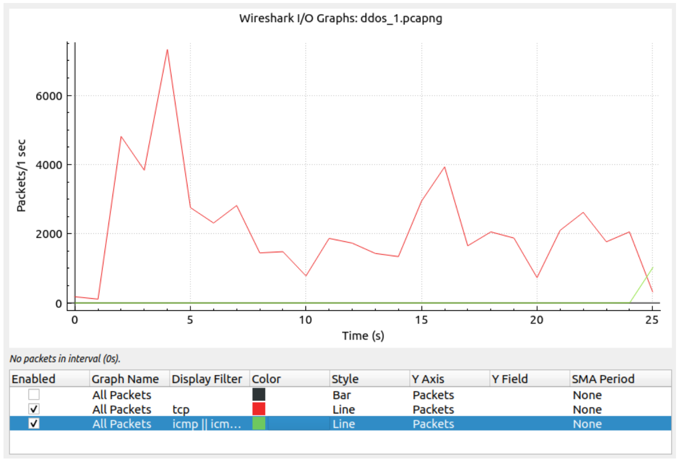
   After applying the filter (ip.addr == 192.168.232.95 && udp) in wireshark, I know that the size of an attack packet is 482 bytes.



Click to select packet 181 (0s = 181).

Enabled	Graph Name	Display Filter	Color	Style
✓	All Packets		⬛	Line
✓	All Packets	ip.addr == 192.168.232.95	🟥	Bar
✓	All Packets	ip.addr==192.168.232.95 && udp	🟦	Bar

Figure 1: find protocol



No packets in interval (0s).

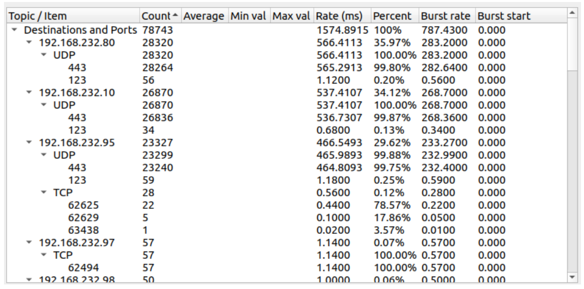
Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
☐	All Packets		⬛	Bar	Packets		None
✓	All Packets	tcp	🟥	Line	Packets		None
✓	All Packets	icmp || icm…	🟩	Line	Packets		None

Figure 2: find intention

3. According to Figure 2, there is a significant decrease in TCP flow after the attack(24s). Thus, we can infer that the victim of this attack is a server and it intended to occupy the transmission

bandwidth. The other evidence is that there is an increase in ICMP flow. It is common after a UDP ddos attack to a server because attackers can send lots of spoofed UDP packets and the server would respond with a ICMP (ping) packet to inform the sender that the destination was unreachable.

4. With Figure 3, we can target the three victims. Then, we can take a closer look at the ip send packets to these victims, whose protocol is UDP to find the amplifiers.



Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent	Burst rate	Burst start
Destinations and Ports	78743				1574.8915	100%	787.4300	0.000
192.168.232.80	28320				566.4113	35.97%	283.2000	0.000
UDP	28320				566.4113	100.00%	283.2000	0.000
443	28264				565.2913	99.80%	282.6400	0.000
123	56				1.1200	0.20%	0.5600	0.000
192.168.232.10	26870				537.4107	34.12%	268.7000	0.000
UDP	26870				537.4107	100.00%	268.7000	0.000
443	26836				536.7307	99.87%	268.3600	0.000
123	34				0.6800	0.13%	0.3400	0.000
192.168.232.95	23327				466.5493	29.62%	233.2700	0.000
UDP	23299				465.9893	99.88%	232.9900	0.000
443	23240				464.8093	99.75%	232.4000	0.000
123	59				1.1800	0.25%	0.5900	0.000
TCP	28				0.5600	0.12%	0.2800	0.000
62625	22				0.4400	78.57%	0.2200	0.000
62629	5				0.1000	17.86%	0.0500	0.000
63438	1				0.0200	3.57%	0.0100	0.000
192.168.232.97	57				1.1400	0.07%	0.5700	0.000
TCP	57				1.1400	100.00%	0.5700	0.000
62494	57				1.1400	100.00%	0.5700	0.000
192.168.232.98	50				1.0000	0.06%	0.5000	0.000

Figure 3: find victim

(1) Victim 1
- IP: 192.168.232.80
- the number of the packets sent to the victim: 28320
- top 3 amplifiers: 28.111.19.188, 129.236.255.89, 124.120.108.157

(2) Victim 2
- IP: 192.168.232.10
- the number of the packets sent to the victim: 26870
- top 3 amplifiers: 4.93.220.190, 5.104.141.250, 124.120.108.157

(3) Victim 3
- IP: 192.168.232.95
- the number of the packets sent to the victim: 23299
- top 3 amplifiers: 34.93.220.190, 128.11.19.188, 212.27.110.13

5. I used the following command to send monlist query.

```
nmap -sU -pU:123 -Pn -n --script=ntp-monlist 142.44.162.188
```

Then, I only focus on the packets, whose protocol is NTP and its source or destination is "142.44.162.188". Finally, I compute the sum of the size of packets I sent and I receive are 76 bytes and 41652 bytes respectively. Thus,

$$\text{amplification factor} = \frac{41652}{76} \approx 548.053.$$

6. (1) amplifier:
Disabling IP routing could be an useful precaution because it forbids a packet from specifying its route. In this way, a packet can no longer manipulate its transmission path, such as BGP

hijacking. Moreover, we can extend this idea to do filter out those packets with spoofing source IP. Although this mechanism may cause some false positive, this trade off is still worthy.

(2) network administrator:

As a network administrator, we can simply implement a filter mechanism, which could detect possible malicious IPs or specific packets. Once our machine identifies a packet or IP as an attack, we could drop the packet, block the IP or put such events in a buffer and keep track of them for fear that we block normal clients. Besides, we can further pay attention to the protocol of packets, such as UDP in the attack in 1.2. We can set a maximum rate for such packets being received and drop the others out of the limit. In this ways, we can prevent from running out of the bandwidth of our server.

Smart Contract

QQ 我來不及寫也不會寫

Web Authentication

a) All code within this problem is written in python with flask package. Besides, I ran them on linux6 workstation, so the host IP is fixed in these files.

(1) Basic HTTP Authentication Scheme (RFC 7617):

- Flag: CNS{H77P_4U7h_r0CK2}
- Implementation: In this mechanism, I simply compare if the username and the password is correct.

(2) Cookie-based Authentication:

- Flag: CNS{CooK135_4R3_d3L1c1ou2}
- Implementation: There are two methods, ”GET” and ”POST”, in this mechanism. First, judge sends a ”GET” request to my server and my server will return a set cookie with the information of the user (e.g. username) if the credential is correct. After that, each time the user wants to connect to my server again, all he has to do is to emit a ”GET” request containing the cookies received from us previously, my server will allow its connection.

(3) JWT-based Authentication:

- Flag: CNS{jwt_5H0UlD_8E_pLacED_1N_8EArer}
- Implementation: Most things is identical to the implementation in cookies. However, my server returns a jwt encoded token instead of a set cookie. Moreover, I used HS256 as the signature algorithm because it is a symmetric encryption, which is more convenient in implementation of JWT.

b) (1) Basic HTTP Authentication Scheme (RFC 7617):

- How it works: In basic HTTP authentication, the client sends a requesta request contains a header field in the form of "Authorization: Basic ¡credentials¿", where credentials is the Base64 encoding of username and password. The server then decodes the credentials, verifies them, and grants access if they are valid.
- Pros: Basic authentication is easy to implement and widely supported by web servers and clients.
- Cons: The credentials are encoded, not encrypted, which means they can be easily decoded if intercepted. This makes basic authentication vulnerable to security risks.

(2) Cookie-based Authentication:

- How it works: After successful authentication, the server generates a session cookie and sends it back to the client. The client has to store the cookie and includes it in subsequent requests. The server can verify the cookie to authenticate the client for each request.
- Pros: The server doesn't need to store session information because all necessary data is stored on the client-side as a cookie, which allows for scalability.
- Cons: If a malicious script gains access to the client's cookies (via XSS attack), it can impersonate the user and perform actions on their behalf, which is known as Cross-Site Scripting (XSS) vulnerabilities.

(3) JWT-based Authentication:

- How it works: After successful login, the server generates a JWT containing the user's claims (e.g., username) and signs it with a secret key. The token is sent to the client, which includes it in subsequent requests. The server verifies the token's signature and extracts the user's claims for authentication.
- Pros: JWTs are digitally signed using either a secret (HMAC) or a public/private key pair (RSA or ECDSA) which safeguards them from being modified by the client or an attacker.
- Cons: Due to their self-contained nature and stateless verification process, it can be difficult to revoke a JWT before it expires naturally. Therefore, actions like banning a user immediately cannot be implemented easily.

c) CNS{JW7_15_N07_a_900d_PLACE_70_H1DE_5ecrE75}

I got this flag by decoding the token hidden in the cookie.

d) CNS{2FA_15_9R347_y0U_5H0Uld_h4v3_0N3}

Generally, I follow the steps as follow to construct TOTP.
Reference website

- Divide the current timestamp by 30. (timer number)
- Encode it as a 64-bit integer
- Write the encoded bytes to a SHA-1 HMAC initialized with the TOTP shared key
- Compute offsets = hmac[-1] & 0xF
- Compute hash = decode hmac[offset   offset + 4] as a 32-bit integer
- Compute (hash & 0x7FFFFFFF) % 1000000

- Fill in 0 from left to right if the length isn't 6

Besides, I use the bulit-in library in python to check my answer when I got wrong results.

e) QQ 我問過了但來不及寫

f) Open Authorization (OAuth)

- How it works: Here is the outline of how OAuth 2.0 works in Figure 4.

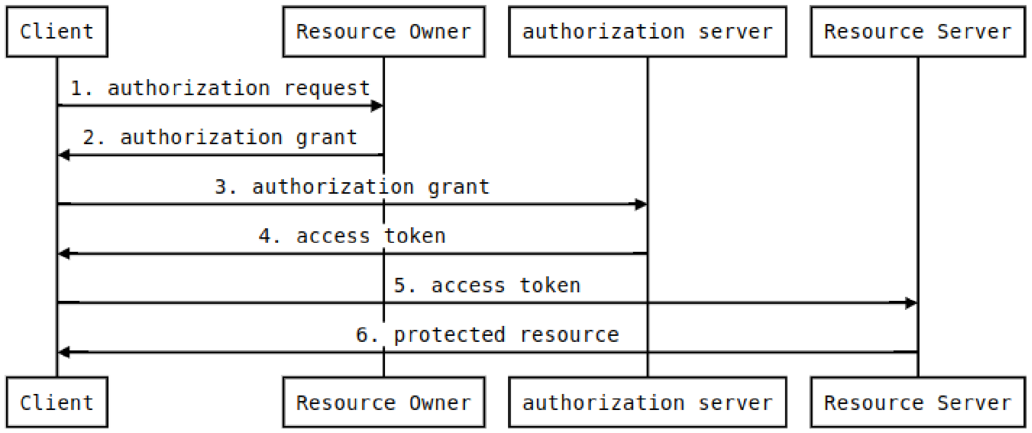


Figure 4: Workflow of OAuth

In this mechanism, clients/users must refresh its token upon it expiring, or they can no longer interact with server.

- Pros: (It outperform the traditional method in these aspects.)
  - User-Friendly: OAuth 2.0 allows users to grant limited access to their resources without sharing their credentials with the client application. This enhances user privacy and security by reducing the risk of credentials being compromised.
  - Scalability: OAuth 2.0 is highly scalable and can be implemented in distributed environments, making it suitable for large-scale applications and platforms.
  - High deployment: OAuth 2.0 is widely adopted and supported by major platforms and services, such as Google, Microsoft, Amazon, and Twitter.
- Cons:
  - Complexity: Implementing OAuth 2.0 can be complex. It involves multiple steps, interactions, and the use of cryptographic techniques, which can significantly increase the complexity.
  - Lack of Uniformity: Auth 2.0 allows flexibility in its implementation, resulting in variations and potential inconsistencies across different platforms. This can make it challenging for developers to ensure interoperability and compatibility between different platform.
  - Token Management: OAuth 2.0 relies heavily on the management and security of access tokens. If access tokens are not properly secured or revoked when necessary, it can lead to potential security vulnerabilities and unauthorized access.

- Security Issues:
  Although OAuth 2.0 has been adopted by many major platforms, there are still some security issues. For example, it doesn't provide a reliable channel for communication between clients and servers. Besides, token management could be a weak link because of its complexity and be prone to attacks such as token hijacking.
  However, the most serious is that OAuth is vulnerable to phishing attacks, where an attacker impersonates a legitimate authorization server or resource server to trick users into granting access to their resources. In fact, users of Gmail has been targeted by an OAuth-based phishing attack in April and May 2017.
- Reference: wiki, stfalcon, chatgpt, Information Security Newspaper

Accumulator

a) Here is the outputs of my code.

```
'accumulatorrrrrr' is in the set.
'QAQ' is not in the set.
```

Here are how I implemented those functions in *accumulator.py*

- Digest: I apply digest = $\text{digest}^{\text{m}_i}\%N$ for all members in accumulator with initial digest = g.
- MembershipProof: To generate a proof for membership, I do similar thing as I did in Digest(). However, I skip the member we want to prove it.
- MembershipVerification: I just check whether $proof^m$ % N equals to $digest$, where m is the hash of the member we want to prove.
- NonMembershipProof: To generate a non-membership proof, I compute the xgcd on m and delta, where delta is the multiplication of all members, and get a and b. Finally, returning the proof, $(g^a\%N, b)$.
- NonMembershipVerification: Let the proof = (ga, b). There are two things to check as follow:

$$\begin{cases} ga^m \cdot digest^b = g^{(a\cdot m+b\cdot delta)} \\ g^{(a\cdot m+b\cdot delta)} = g \end{cases}$$

  Only when the two equations are right, I return true in this function.

b) cns{ph4k3_m3m83r5H1p!}

ref1 Since we know the value of $\phi(N)$, we can simply compute the multiplicative inverse(k) of any message(m) we send. Next, I send $digest^k$ % N as the membership proof. In this way, we can make sure that

$$proof^m = (digest^k)^m = digest(\text{mod}N),$$

which can pass the verification.

c) cns{N0N_n0n_m3M83RSh1p!}

Since we know the value of $\phi(N)$, we can simply let $b = 0$ and a be the multiplicative inverse of the message we send.