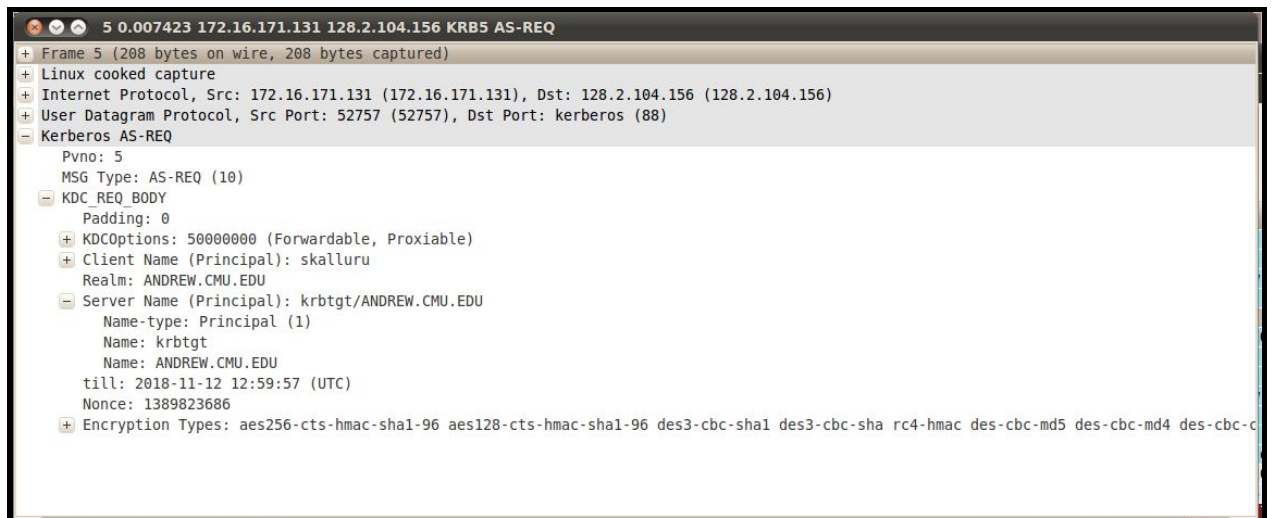


18-631 INFORMATION SECURITY HOME WORK 3

SAI VINEETH KALLURU SRINIVAS

QUESTION 1 - KERBEROS WIRESHARK

- 1) Step 6 is optional, because Server can certify only if it wishes to or not. Because the Authenticator 'C' is encrypted with shared key between C(client) and Server (V). A Server (V) should not be able to decrypt the ticket unless it has the key associated with the Kerberos principal. If it can't decrypt the service ticket then it shouldn't have access to the information about the Client (C) which it needs to properly authenticate and create a session for that client. Therefore the mere fact that the server is able to construct a reply to C is itself sufficient to say that the right server has responded.
- 2) 2.1) I am running Kerberos Version 5 (Pvno : 5) as shown by the packet below

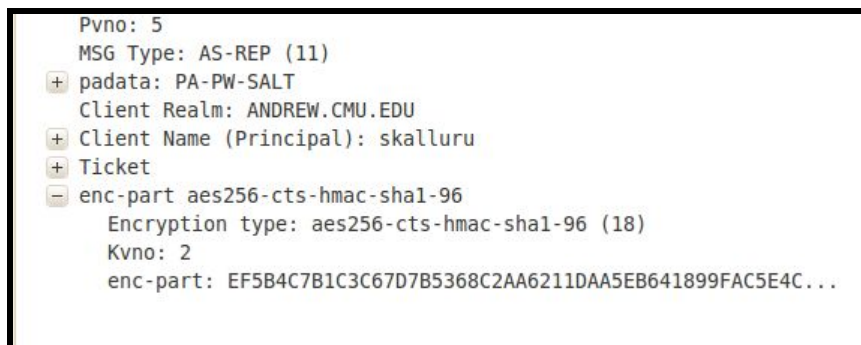


2.2) Server Name : **krbtgt** , ANDREW.CMU.EDU

2.3) Client Name : skalluru (my andrew ID)

2.4) Encryption type used : aes256 - cts - hmac - sha1 - 96

2.5) Encrypted part: EC5B4C7B (first 8 letters of encrypted part)



3.1) Output of KLIST 2

```
user@netsec-hw:~$ klist
Credentials cache: FILE:/tmp/krb5cc_1000
Principal: skalluru@ANDREW.CMU.EDU

    Issued            Expires            Principal
Nov 11 21:59:57  Nov 12 07:59:57  krbtgt/ANDREW.CMU.EDU@ANDREW.CMU.EDU
user@netsec-hw:~$
```

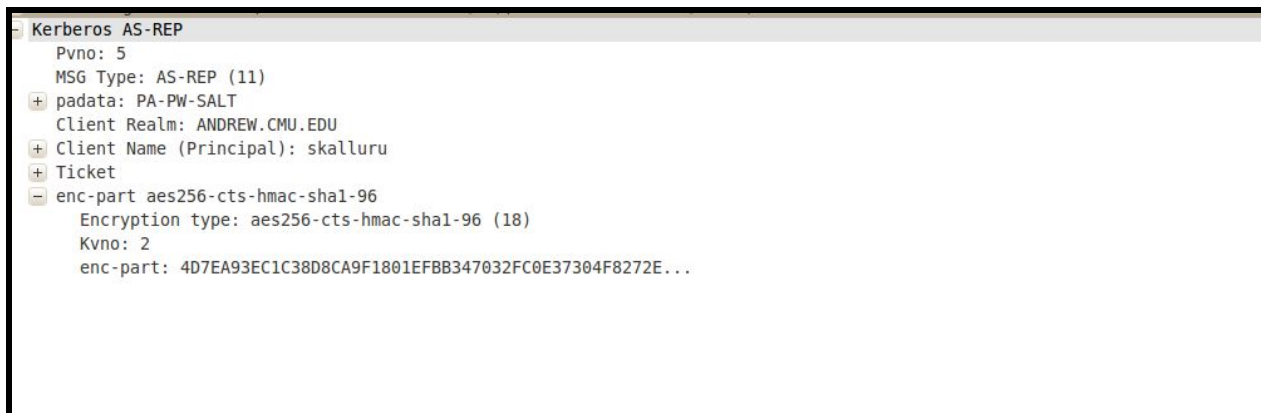
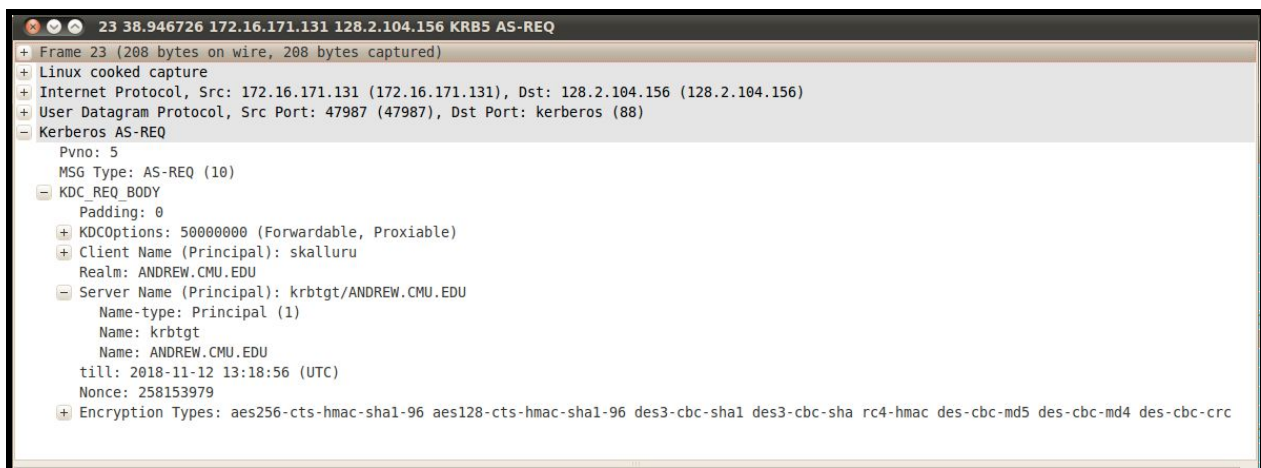
3.2) Currently only one ticket is valid

3.3) It is valid for 10 hours (NOV11 9PM - NOV12 8AM)

3.4) principals involved are : **skalluru** and

krbtgt/ANDREW.CMU.EDU@ANDREW.CMU.EDU

4.1) The four packets of AS REQ, AS REP, TGS REQ and TGS REP are shown below



```
+ Frame 35 (617 bytes on wire, 617 bytes captured)
+ Linux cooked capture
+ Internet Protocol, Src: 172.16.171.131 (172.16.171.131), Dst: 128.2.104.156 (128.2.104.156)
+ User Datagram Protocol, Src Port: 40691 (40691), Dst Port: kerberos (88)
- Kerberos TGS-REQ
  Pvnno: 5
  MSG Type: TGS-REQ (12)
  + padata: PA-TGS-REQ
  - KDC_REQ_BODY
    Padding: 0
    + KDCOptions: 00010000 (Canonicalize)
    Realm: ANDREW.CMU.EDU
    - Server Name (Principal): afs
      Name-type: Principal (1)
      Name: afs
    till: 1970-01-01 00:00:00 (UTC)
    Nonce: 1757861563
    + Encryption Types: des-cbc-crc
```

```
36 39.222635 128.2.104.156 172.16.171.131 KRB5 TGS-REP
+ Frame 36 (562 bytes on wire, 562 bytes captured)
+ Linux cooked capture
+ Internet Protocol, Src: 128.2.104.156 (128.2.104.156), Dst: 172.16.171.131 (172.16.171.131)
+ User Datagram Protocol, Src Port: kerberos (88), Dst Port: 40691 (40691)
- Kerberos TGS-REP
  Pvnno: 5
  MSG Type: TGS-REP (13)
  Client Realm: ANDREW.CMU.EDU
  + Client Name (Principal): skalluru
  + Ticket
  - enc-part aes256-cts-hmac-sha1-96
    Encryption type: aes256-cts-hmac-sha1-96 (18)
    enc-part: F122B272085FD11C4AC4A3D3F723B2EC58AD0D5DE388F29C...
```

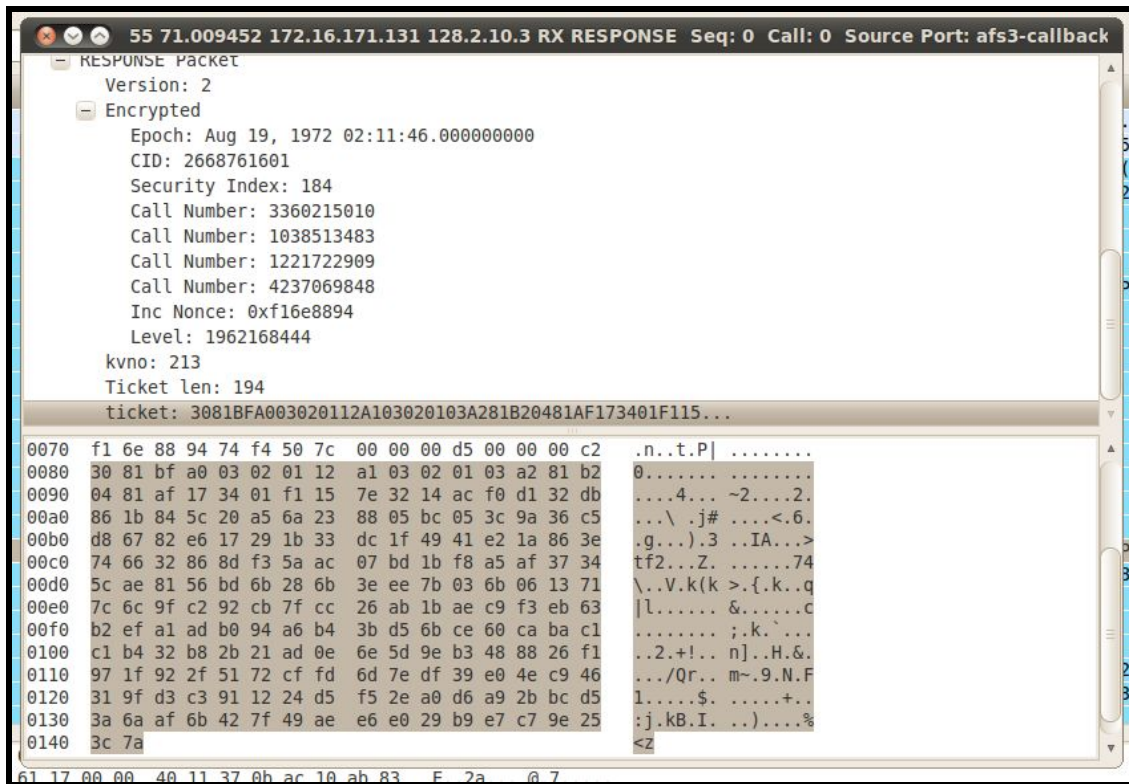
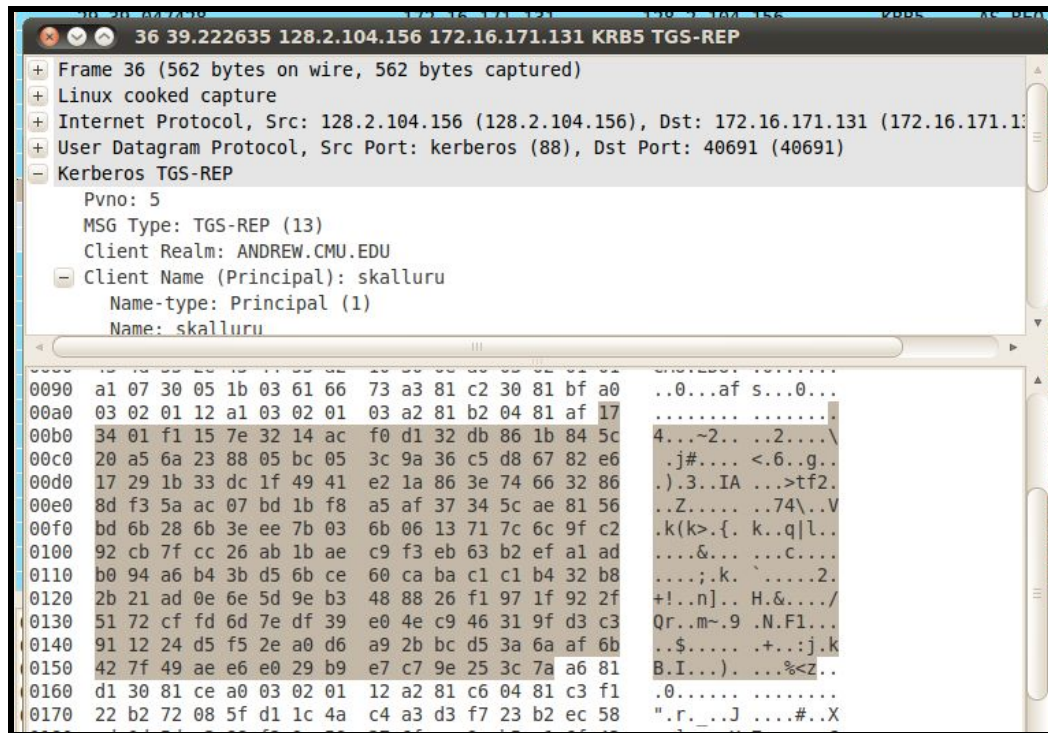
4.2) The name of the server can be found in TGS REQ as the ID_V of the server is sent in plain text in request to Ticket granting server. The name of the server is : ANDREW FILE SYSTEM - **afs**

```
+ User Datagram Protocol, Src Port: 40691 (40691), Dst Port: kerberos (88)
- Kerberos TGS-REQ
  Pvnno: 5
  MSG Type: TGS-REQ (12)
  - padata: PA-TGS-REQ
    - Type: PA-TGS-REQ (1)
      Value: 6E8201BB308201B7A003020105A10302010EA20703050000... AP-REQ
        Pvnno: 5
        MSG Type: AP-REQ (14)
        Padding: 0
        + APOptions: 00000000
        - Ticket
          Tkt-vno: 5
          Realm: ANDREW.CMU.EDU
          - Server Name (Principal): krbtgt/ANDREW.CMU.EDU
            Name-type: Principal (1)
            Name: krbtgt
            Name: ANDREW.CMU.EDU
          + enc-part aes256-cts-hmac-sha1-96
            + Authenticator aes256-cts-hmac-sha1-96
        - KDC_REQ_BODY
          Padding: 0
```

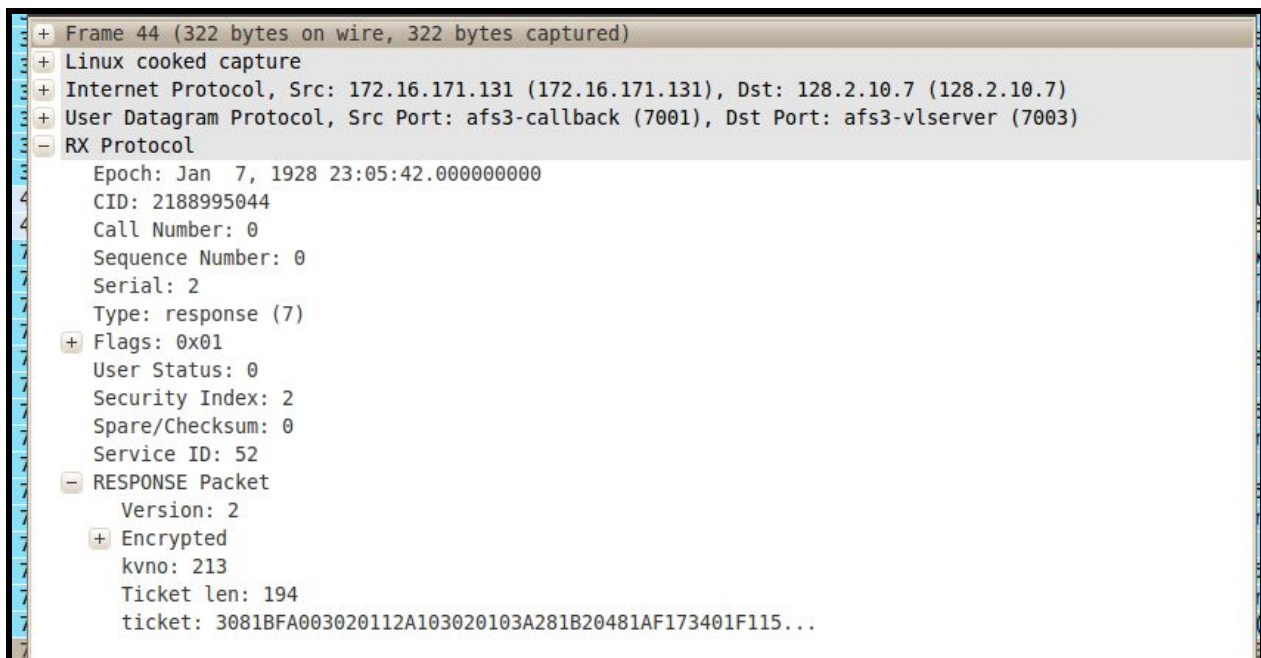
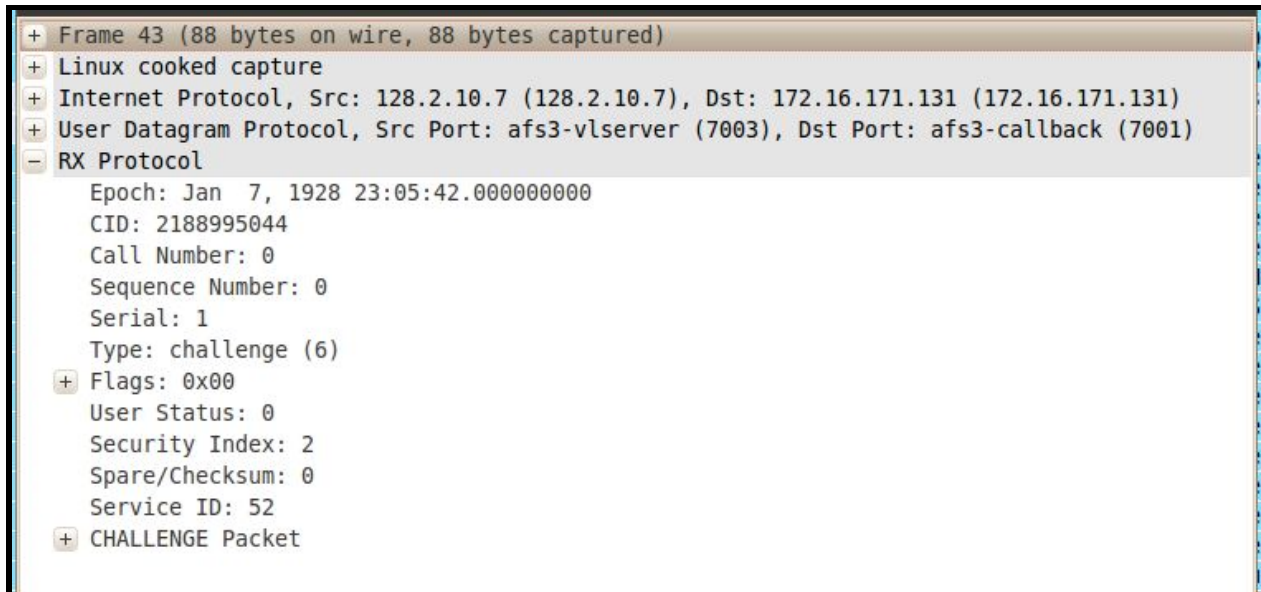
4.3) From TGS-REQ , we can see there is Ticket block being sent in the packet, which proves that the ticket from Authentication Server is sent to TGS - Ticket Granting server.

4.4) We can see from the ticket given from TGS - REP is given by client to server.

The value of the ticket starts from 173401f1157e32 in TGS REP (line 00a0 to 0150), which is also available in RX response (line 0090 to 0140 ending with 25 3c 7a).



4.5) Mutual Authentication is used, because we can see an RX CHALLENGE and an RX RESPONSE packet. Which implies that the client is asking AFS to authenticate itself, for which the server responds to that challenge. Frame 43 - RX - CHALLENGE packet
Frame 44 - RX RESPONSE packet.



5.1)

```
user@netsec-hw:~$ klist
Credentials cache: FILE:/tmp/krb5cc_1000
Principal: skalluru@ANDREW.CMU.EDU

    Issued            Expires            Principal
Nov 11 22:18:56  Nov 12 08:18:56  krbtgt/ANDREW.CMU.EDU@ANDREW.CMU.EDU
Nov 11 22:18:56  Nov 12 08:18:56  afs@ANDREW.CMU.EDU
```

5.2) Two tickets are valid

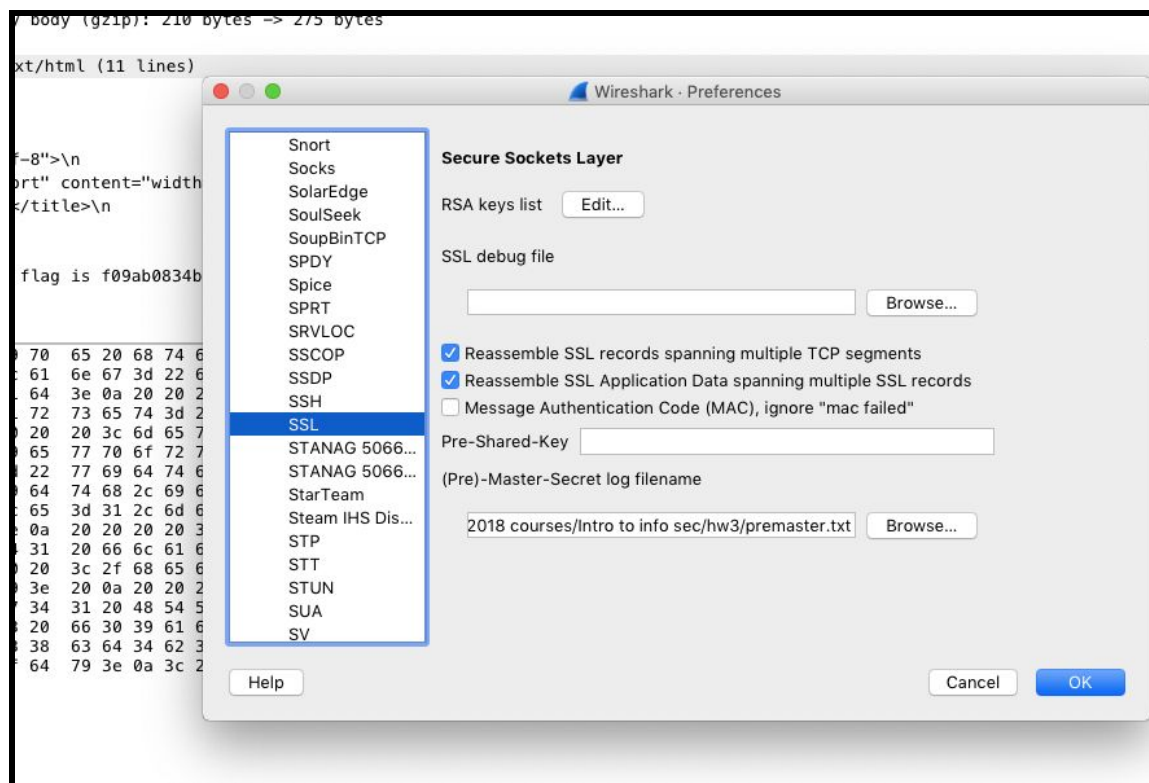
5.3) Both of them expire in 12 hours

5.4) Three principals are involved:

- skalluru
- krbtgt/[ANDREW.CMU.EDU@ANDREW.CMU.EDU](#)
- afs@ANDREW.CMU.EDU

2. DECRYPTING SSL TRAFFIC

- SSL Traffic is captured in ***picocft ssl flag1.pcapng*** file. The traffic capture can be parsed using Wireshark.
- Once we open the capture in wireshark, we can see red lines, which shows the encrypted part of the traffic. We have to decrypt this traffic to see the entire SSL handshake.
- The SSL traffic can be decrypted by using premaster.txt log file. This log file is generated by browser, (if set to TRUE) during SSL sessions. The log file contains keys that could decrypt the SSL traffic.
- We can add the premaster.txt key log file in the Wireshark to decrypt the traffic.
- The procedure to add premaster.txt log file is as follows
 - First, we should check if wireshark version is latest.
 - Second we should add a system variable named SSLKEYLOGFILE having the value as location to premaster.txt (ex : C:\temp\premaster.txt)
 - We should then go to edit preferences menu in Wireshark, and choose SSL in protocols section as shown below



- In the menu, we can edit the path to premaster secret log filename and its path.

- The Wireshark will use this premaster.txt file and decrypt the traffic as shown below

Wireshark packet capture showing network traffic. The interface is set to 'Narrow & Wide' view. The packet list on the left shows packets 1 through 24. The packet details pane on the right shows the selected packet (No. 13) with details for the TLSv1.2 Client Hello message. The packet bytes pane at the bottom shows the raw data of the selected packet.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	128.2.16.49	52.14.175.183	TCP	66	58538 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.000275	128.2.16.49	52.14.175.183	TCP	66	58539 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.000533	128.2.16.49	52.14.175.183	TCP	66	58540 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
4	0.000754	128.2.16.49	52.14.175.183	TCP	66	58541 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
5	0.000974	128.2.16.49	52.14.175.183	TCP	66	58542 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
6	0.001185	128.2.16.49	52.14.175.183	TCP	66	58543 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
7	0.031957	52.14.175.183	128.2.16.49	TCP	66	443 → 58538 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
8	0.032079	128.2.16.49	52.14.175.183	TCP	54	58538 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
9	0.032104	52.14.175.183	128.2.16.49	TCP	66	443 → 58539 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
10	0.032143	128.2.16.49	52.14.175.183	TCP	54	58539 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
11	0.032189	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello
12	0.032356	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello
13	0.032423	52.14.175.183	128.2.16.49	TCP	66	443 → 58540 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
14	0.032463	128.2.16.49	52.14.175.183	TCP	54	58540 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
15	0.032504	52.14.175.183	128.2.16.49	TCP	66	443 → 58541 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
16	0.032555	128.2.16.49	52.14.175.183	TCP	54	58541 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
17	0.032567	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello
18	0.032686	52.14.175.183	128.2.16.49	TCP	66	443 → 58542 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
19	0.032756	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello
20	0.032790	128.2.16.49	52.14.175.183	TCP	54	58542 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
21	0.032931	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello
22	0.032967	52.14.175.183	128.2.16.49	TCP	66	443 → 58543 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
23	0.033012	128.2.16.49	52.14.175.183	TCP	54	58543 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
24	0.033127	128.2.16.49	52.14.175.183	TLSv1.2	253	Client Hello

- The dark grey lines show the traffic that is decrypted.
- We can now clearly see and distinguish between client hello, server hello and all other handshakes between them.
- To find the flag, we have to look for an HTML handshake between server and client, which was discovered :

Wireshark packet capture showing network traffic. The interface is set to 'Narrow & Wide' view. The packet list on the left shows packets 72 through 85. The packet details pane on the right shows the selected packet (No. 79) with details for the HTTP GET request. The packet bytes pane at the bottom shows the raw data of the selected packet.

No.	Time	Source	Destination	Protocol	Length	Info
72	0.070816	52.14.175.183	128.2.16.49	TLSv1.2	1514	Certificate [TCP segment of a reassembled PDU]
73	0.070818	52.14.175.183	128.2.16.49	TLSv1.2	342	Server Key Exchange, Server Hello Done
74	0.070842	128.2.16.49	52.14.175.183	TCP	54	58543 → 443 [ACK] Seq=200 Ack=5845 Win=65536 Len=0
75	0.074510	128.2.16.49	52.14.175.183	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Finished
76	0.077271	128.2.16.49	52.14.175.183	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Finished
77	0.079236	128.2.16.49	52.14.175.183	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Finished
78	0.080601	128.2.16.49	52.14.175.183	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Finished
79	0.081664	128.2.16.49	52.14.175.183	HTTP	640	GET /14741.html HTTP/1.1
80	0.099558	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished
81	0.101979	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished
82	0.106913	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished
83	0.109489	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished
84	0.111425	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished
85	0.113071	52.14.175.183	128.2.16.49	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Finished

- The HTML GET request has the packet details as follows:

Wireshark packet details pane showing the details of the selected packet (No. 79). The details are organized into sections: Transmission Control Protocol, Secure Sockets Layer, Hypertext Transfer Protocol, and Cookie. The Hypertext Transfer Protocol section shows the GET request details, including the Host, Connection, Upgrade-Insecure-Requests, User-Agent, Accept, Accept-Encoding, Accept-Language, Cookie, and If-None-Match headers. The Cookie section shows the cookie details, including the _ga and _gid cookies.

Section	Field	Value
Transmission Control Protocol	Src Port	58538
	Dst Port	443
Secure Sockets Layer	Seq	326
	Ack	5845
Hypertext Transfer Protocol	GET	/14741.html HTTP/1.1
	Host	picocf.com
	Connection	keep-alive
	Upgrade-Insecure-Requests	1
	User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.101 Safari/537.36
	Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
	Accept-Encoding	gzip, deflate, br
	Accept-Language	en-US,en;q=0.8
	Cookie	_ga=GA1.2.209658423.1500901750; _gid=GA1.2.47767360.1503325306
	If-None-Match	"113-5574461e05f9e-gzip"
Cookie	_ga	GA1.2.209658423.1500901750
	_gid	GA1.2.47767360.1503325306

- The packet details say that, the response to the packet is in Frame : 86.
Therefore, if we open frame 86 and expand the HTML inline text , we can see the flag

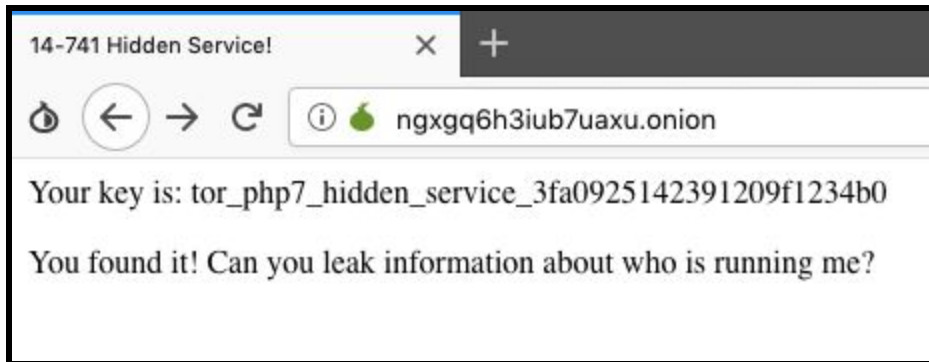
```

[HTTP response 1/2]
[Time since request: 0.032368000 seconds]
[Request in frame: 79]
[Next request in frame: 93]
[Next response in frame: 94]
Content-encoded entity body (gzip): 210 bytes -> 275 bytes
File Data: 275 bytes
▼ Line-based text data: text/html (11 lines)
<!doctype html>\n
<html lang="en">\n
<head>\n
  <meta charset="utf-8">\n
  <meta name="viewport" content="width=device-width,initial-scale=1,minimal-ui">\n
  <title>14741 flag</title>\n
</head>\n
<body> \n
  The 14741 HTTPS flag is f09ab0834bfa0238cd4b54aa\n
</body>\n
</html>\n
0000 3c 21 64 6f 63 74 79 70 65 20 68 74 6d 6c 3e 0a <!doctype e html>\n
0010 3c 68 74 6d 6c 20 6c 61 6e 67 3d 22 65 6e 22 3e <html la ng="en">\n
0020 0a 20 20 3c 68 65 61 64 3e 0a 20 20 20 20 3c 6d <head >\n
0030 65 74 61 20 63 68 61 72 73 65 74 3d 22 75 74 66 <meta char set="utf-8">\n
0040 2d 38 22 3e 0a 20 20 20 20 3c 6d 65 74 61 20 6e <meta n ame="vie wport" c ontent=" width=de vice-wid th,initi al-scale =1,minim al-ui">\n
0050 61 6d 65 3d 22 76 69 65 77 70 6f 72 74 22 20 63 <tit le>14741 flag</t itle>\n
0060 6f 6e 74 65 6e 74 3d 22 77 69 64 74 68 2c 69 6e 69 74 69 </head>\n
0070 76 69 63 65 2d 77 69 64 74 68 2c 69 6e 69 74 69 <body> \n
0080 61 6c 2d 73 63 61 6c 65 3d 31 2c 6d 69 6e 69 6d The 1474 1 HTTPS
0090 61 6c 2d 75 69 22 3e 0a 20 20 20 3c 74 69 74 flag is f09ab083
00a0 6c 65 3e 31 34 37 34 31 20 66 6c 61 67 3c 2f 74 4bf0238 cd4b54aa
00b0 69 74 6c 65 3e 0a 20 20 3c 2f 68 65 61 64 3e 0a </bod y> </htm l>\n
00c0 20 20 3c 62 6f 64 79 3e 20 0a 20 20 20 20 20 20 \n
00d0 54 68 65 20 31 34 37 34 31 20 48 54 54 50 53 20 \n
00e0 66 6c 61 67 20 69 73 20 66 30 39 61 62 30 38 33 \n
00f0 34 62 66 61 30 32 33 38 63 64 34 62 35 34 61 61 \n
0100 0a 20 20 3c 2f 62 6f 64 79 3e 0a 3c 2f 68 74 6d </bod y> </htm l>\n
0110 6f 3e 0a \n
  
```

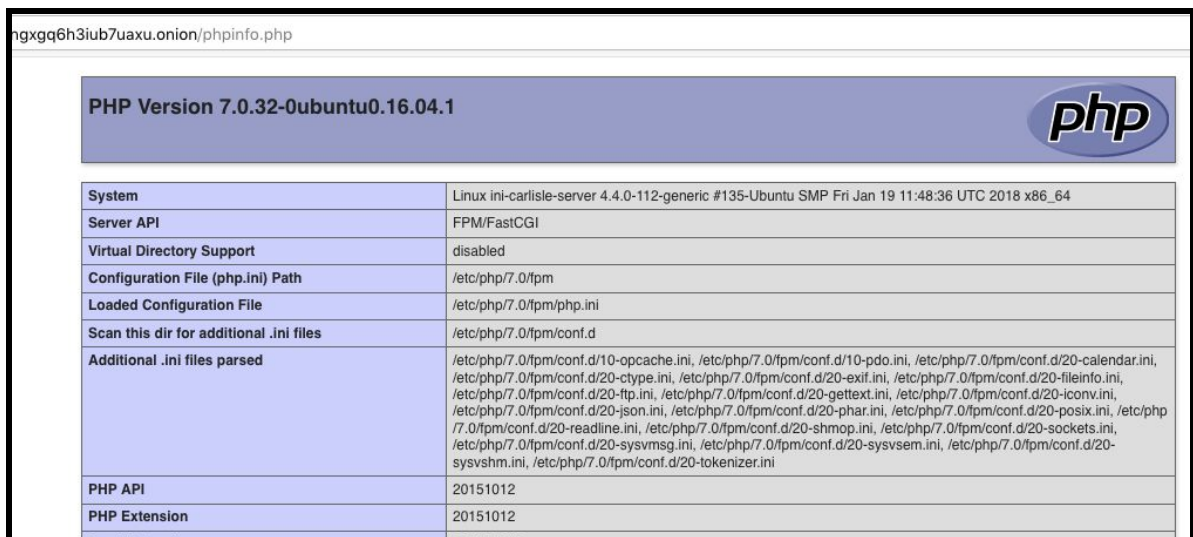
- In this way we can decrypt the SSL traffic and get the data sent between server and client, given that we have session logs saved by the browser.
- Please zoom into the pictures for letters to become more legible.
- The flag is found to be : **f09ab0834bfa0238cd4b54aa**
- The environmental variable used for chrome is : SSLKEYLOGFILE
- HTTPS is secure, even though it can be decrypted because, we need the private keys to decrypt this traffic, and these keys are **available only at the client side** browser that requested for the handshake, and will not be available with the man in the middle.
- HTTPS adds security and trust as the MiTM cannot alter the packet while it is in transit.

3. TOR HIDDEN SERVICE

- Tor - is an online anonymity services provider.
- Tor hidden services are used by individuals or organisations to run servers that are not easier to track (provides anonymity to server information, hostname, and other details)
- However if the service is not configured properly, The service might be vulnerable to spill some valuable information about the server to any hacker.



- From this picture we can see that, the key is being spilled out by *onion* service. In the key, we get information about the kind of service it is using - *php* service.
- Because of this leak of information we have come to know that we can use - ***php information disclosure vulnerability*** - which would provide valuable information about the server.
- Just by changing the URL as - <http://ngxgq6h3iub7uaxu.onion/phpinfo.php> , the part in bold can be added to the onion service URL and we can get information about the server as shown below.



PHP Version 7.0.32-0ubuntu0.16.04.1	
System	Linux ini-carlisle-server 4.4.0-112-generic #135-Ubuntu SMP Fri Jan 19 11:48:36 UTC 2018 x86_64
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php/7.0/fpm
Loaded Configuration File	/etc/php/7.0/fpm/php.ini
Scan this dir for additional .ini files	/etc/php/7.0/fpm/conf.d
Additional .ini files parsed	/etc/php/7.0/fpm/conf.d/10-opcache.ini, /etc/php/7.0/fpm/conf.d/10-pdo.ini, /etc/php/7.0/fpm/conf.d/20-calendar.ini, /etc/php/7.0/fpm/conf.d/20-ctype.ini, /etc/php/7.0/fpm/conf.d/20-exif.ini, /etc/php/7.0/fpm/conf.d/20-fileinfo.ini, /etc/php/7.0/fpm/conf.d/20-ftp.ini, /etc/php/7.0/fpm/conf.d/20-gettext.ini, /etc/php/7.0/fpm/conf.d/20-iconv.ini, /etc/php/7.0/fpm/conf.d/20-json.ini, /etc/php/7.0/fpm/conf.d/20-phar.ini, /etc/php/7.0/fpm/conf.d/20-posix.ini, /etc/php/7.0/fpm/conf.d/20-readline.ini, /etc/php/7.0/fpm/conf.d/20-shmop.ini, /etc/php/7.0/fpm/conf.d/20-sockets.ini, /etc/php/7.0/fpm/conf.d/20-sysvmsg.ini, /etc/php/7.0/fpm/conf.d/20-sysvsem.ini, /etc/php/7.0/fpm/conf.d/20-sysvshm.ini, /etc/php/7.0/fpm/conf.d/20-tokenizer.ini
PHP API	20151012
PHP Extension	20151012

- From this page, a hacker can get most information about the server and the services enabled on the server, which could be valuable for bringing down an attack.
- We can see that the server is **Linux Ubuntu 0.16.04.1**
- We can also see that the host name is - **ini-carlisle-server**. Therefore **no longer anonymous**.

4. ALICE AND BOB - PART 1

1. If attacker gets K'_{AB} , the attacker would not be able to read messages $M1$ and $M2$, because Hash functions are NOT REVERSIBLE FUNCTIONS. As in, knowing the output we cannot determine the input to the function. In order to determine messages $M1$ and $M2$ we need key K , since K is not obtainable from K'_{AB} we can safely conclude that all messages from previous sessions **cannot be decrypted**. The previous session **remains a secret**.
2. However, if the attacker obtains K'_{AB} , he can compute the $H(K'_{AB})$ because the Hash function is **publicly available**. Therefore if he gets hold of one session key, Attacker can pretty much decoded the hash of the available key and use it to decrypt the messages in future sessions. The future session **does not remain a secret anymore**.
3. In this protocol Alice and Bob, use diffie Hellman key exchange. However they are encrypted using K_{AB} . Getting hold of K_{AB} does not allow the hacker to deduce messages in previous sessions. The messages in previous sessions are encrypted with Key K which is derived from diffie Hellman exchange and not from K_{AB} itself. Having K_{AB} would give access to $g^a \text{ mod } p$ and $g^b \text{ mod } p$, however key $K = g^{(ab)} \text{ mod } p$ is possible only when the attacker get either a or b separately. Since $g^x \text{ mod } p$ is not a reversible function the attacker would not be able to get either random number ' a ' or random number ' b '.
4. The same explanation as earlier would be the reasoning to state that the attacker cannot decrypt messages of both **previous and future sessions** (because the key that **encodes the message - M** for future sessions is not **derived** from past sessions).
5. **COMMENTS : Answers for questions 4.3 and 4.4**, assumes that there is **no Man in the Middle attack** and it also assumes that the **random number a and b are not predictable**. If either of these assumptions becomes true, then this protocol would fail to keep the message a secret. In comparison to protocol shown in subdivision 1 and 2, the protocol in subdivision 3 and 4 performs better as it protects both past and future sessions.

5. ALICE AND BOB - PART 2

- Security properties ensured in Bob's protocol :
 - Confidentiality / Secrecy is enforced - because the message can be decrypted only by using Bob's private key
 - Integrity - the data if tampered with, will tamper the data of the sender name as well. Therefore the validity of the data will not exist and therefore Bob would discard.
- Alice made a mistake by removing the sender's name in her protocol, In Alice's protocol data integrity will not be ensured. Because if the data X is tampered with, then there is no way to determine if it is a valid message or not. Whereas in Bob's protocol if the data is tampered with, sender's name also gets tampered and therefore the receiver can understand that the data is tampered and discard it. Therefore Alice is **wrong** in assuming that Bob's protocol comes down to just sending the message alone.
- The protocol can be enhanced by using **timestamp** along with the sender name and message. The receiver can discard the message when the timestamp is far in the past, above a threshold time value (t secs) . The threshold time value(t secs) can be set by choosing the value such that, it would be less time for Mallory / attacker to replay a packet within that time.
- Even for freshness - **Timestamps** can be used :
 - A → B : ENC :{A,X,T_A} using K_AB
 - After some threshold time t, if Mallory (M) replays the same packet (**ENC :{A,X,T_A} using K_AB**) , then B will discard the packet because, the valid timestamp at that instance would be T_B and not T_A.
 - In other words ,if Mallory replays the same packet again ,timestamp T_A would have expired.
 - Timestamps are *nonces* . Nonces can be used to bring freshness into protocol and prevent replay attacks.
- Nonces should not be predictable - However time stamps could be predictable, therefore there is a need to find NON PREDICTABLE NONCES, for Mallory not to create a new packet with a predictable nonce and impersonate Alice to Bob.

6. SQL INJECTION CTF USERNAME - kssaivineeth15

6. For solving the SQL Injection, we can follow the following steps

- Right click and open - view page source - to view the HTML of the page source
- Click on the PHP page with <a href> tag in the HTML page
- The php page looks like below

```
1 <?php
2 include "config.php";
3 $con = new SQLite3($database_file);
4
5 $username = $_POST["username"];
6 $password = $_POST["password"];
7 $debug = $_POST["debug"];
8 $dog = $_POST["dog"];
9 $query = "SELECT * FROM users WHERE name='$username' AND password='$password'";
10 $result = $con->query($query);
11 if (intval($debug)) {
12     echo "<pre>";
13     echo "username: ", htmlspecialchars($username), "\n";
14     echo "password: ", htmlspecialchars($password), "\n";
15     echo "dog: ", htmlspecialchars($dog), "\n";
16     echo "SQL query: ", htmlspecialchars($query), "\n";
17     echo "</pre>";
18 }
19
20 $row = $result->fetchArray();
21
22 if ($row && (strcasecmp($dog, 'scotty') == 0)) {
23     echo "<h1>Logged in!</h1>";
24     echo "<p>Your flag is: $FLAG</p>";
25 } else {
26     echo "<h1>Login failed.</h1>";
27 }
28 ?>
29
```

-
- The query variable in PHP script handles the SQL query to the database.
- This query is prone to SQL injection
- The username and password field values from the HTML page is directly added into the variables \$username and \$password
- We can also notice that if the *dog* field is given as 'scotty' we can ensure that the **if** condition can become true execute.
- Therefore we can see that the dog field variable has to be 'scotty'
- For the username and password fields, we need not find out the username or password, instead we can inject SQL code such that the \$query turns out to be a valid SQL query.
- To make it valid we can inject multiple varieties of SQL code. I chose
- **'OR '1'='1'** for both username and password fields.
- This injection of SQL code changes the \$query to SELECT * FROM users WHERE name = '1' **OR** '1' = '1' AND password = '1' **OR** '1' = '1'
- We can see that even though there is no name with '1' value or password with '1' value, the OR statement in both cases will make either side of AND as TRUE, this will make the entire SQL statement TRUE.
- This code will make the **if** condition become true and allow me to login and print the FLAG.

7. BLIND SQL INJECTION CTF USERNAME - kssaivineeth15

7. For solving the SQL Injection, we can follow the following steps

- Right click and open - view page source - to view the HTML of the page source
- Click on the PHP page with <a href> tag in the HTML page
- The login.php page looks like below

```
1 <?php
2 include "config.php";
3 $con = new SQLite3($database_file);
4
5 $username = $_POST["username"];
6 $password = $_POST["password"];
7 $debug = $_POST["debug"];
8 $dog = $_POST["dog"];
9 $query = $con->prepare('SELECT * FROM users WHERE name=:username AND password=:password AND dog=:dog');
10 $query->bindValue(':username',$username,SQLITE3_TEXT);
11 $query->bindValue(':password',$password,SQLITE3_TEXT);
12 $query->bindValue(':dog',$dog,SQLITE3_TEXT);
13 $result = $query->execute();
14 if (intval($debug)) {
15     echo "<pre>";
16     echo "username: ", htmlspecialchars($username), "\n";
17     echo "password: ", htmlspecialchars($password), "\n";
18     echo "dog: ", htmlspecialchars($dog), "\n";
19     echo "</pre>";
20 }
21
22 $row = $result->fetchArray();
23
24 if ($row == 0) {
25     echo "<h1>Logged in!</h1>";
26     echo "<p>Your flag is: $FLAG</p>";
27 } else {
28     echo "<h1>Login failed.</h1>";
29 }
30 ?>
31
```

- In login.php the fields \$username and \$password are NOT INJECTABLE as they are passed as SQL PARAMETERS, which get executed only at the runtime of the query. Once they are binded as parameters it is impossible to inject SQL code, as the variables \$username and \$password are treated as values AND NOT AS A PART OF THE SQL QUERY STRING.
- However we provided with another html page - game.html which as game.php as shown below.

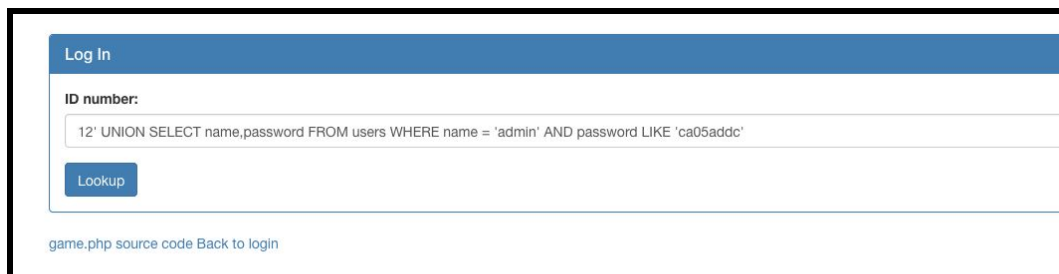
```

1  <?php
2      include "config.php";
3      $con = new SQLite3($database_file);
4
5      $id = $_POST["id"];
6      $debug = $_POST["debug"];
7      $query = "SELECT * FROM games WHERE ID='$id'";
8      $result = $con->query($query);
9      if (intval($debug)) {
10         echo "<pre>";
11         echo "id: ", htmlspecialchars($id), "\n";
12         echo "</pre>";
13     }
14
15     $row = $result->fetchArray();
16
17     if ($row) {
18         echo "<h1>I know that game!</h1>";
19     } else {
20         echo "<h1>What game is that?</h1>";
21     }
22 ?>
23

```

-
- In game.php we can see that both login.php and game.php use the same database file. Also the SQL query in games.php is NOT BINDED AS SQL PARAMETER hence VULNERABLE for SQL INJECTION ATTACK.
- Here the SQL query asks for an ID value from *games* table. However since both *users* and *games* table are present in the same database. We can access *users* table by injecting SQL code into the *games* table query.
- We can achieve this using UNION statement in SQL .
- On checking the value of 1 for ID, we can see that the webpage displays - ' I KNOW THAT GAME' which implies that the SQL query is a valid SQL query. Otherwise it displays ' WHAT GAME IS THAT?'
- This gives us a confirmation that, we cannot directly see the values in the users table, however given a set of conditions we can make this a TRUE/ FALSE problem and try to hit a valid sql query.
- We are given in the hint that the username is 'admin' and the dog is 'scotty' therefore our only concern is with password.
- A general UNION statement in SQL looks like below:
 - SELECT _____ FROM games WHERE ID = ' 1 ' **UNION SELECT _____ FROM users -- '**
 - From the above query, the dashes represent the number of columns required from the selected tables, for UNION statement to work, both sides of UNION statement has to request for equal number of tables.
 - UNION concatenates the output from both queries and gives the union of the output - or in other words makes the SQL query TRUE.
 - We can keep changing the other side of the query and check its validity by check if the UNION statement becomes TRUE.
 - We are interested in *name* and *password* columns of *users* table, therefore we should request for two columns from *games* table as well.
 - To check the number of columns in games table we can try out the following query :

- **SELECT * FROM games WHERE ID = ' 1 ' ORDER BY 2 -- '**
- Order by helps us in finding out the maximum number of columns available in *games* table. As long as the SQL query is TRUE, the column value can be increased, once it becomes FALSE, the corresponding value would be the maximum set of columns in the table
 - Ex if ORDER BY 1 is TRUE and ORDER BY 2 is TRUE and ORDER BY 3 is false, then maximum number of columns in *games* table is 2.
- Therefore we should request for two tables in *users* table as well.
- **SELECT ID, FROM games WHERE ID = ' 1 ' UNION SELECT name, password FROM users -- '** - This change to query will make it TRUE.
- However, we also know that *name* value is 'admin' therefore adding the above information we get
- **SELECT ID, FROM games WHERE ID = ' 1 ' UNION SELECT name, password FROM users WHERE name = 'admin'-- '**
- We can brute force for password by using the following SQL statement.
 - **SELECT ID, FROM games WHERE ID = ' 1 ' UNION SELECT name, password FROM users WHERE name = 'admin' AND password LIKE 'A%'-- ' .**
 - The above SQL query returns TRUE if there is a user named admin and their password starts with the value 'A'. We can use this information and keep brute forcing values of all alphabets and numbers until we hit ' I KNOW THAT GAME' response from the webpage. (However based on question hint, we need to bruteforce only hex characters **0-9 , a-f**)
 - Every time we get a value we can update the SQL statement with the value found it and brute force the rest of characters.
 - **SELECT ID, FROM games WHERE ID = ' 1 ' UNION SELECT name, password FROM users WHERE name = 'admin' AND password LIKE 'c%'**



-
- The password after brute forcing was found to be - ca05addc (Please zoom into for the letters in the image to become legible).
- Adding the name = 'admin' and password = 'ca05addc' and dog = 'scotty'. we get logged in and claim the flag.


```
SELECT ID FROM USERS WHERE USERNAME = 'ADMIN' AND SUBSTR(PASSWORD,1,1) == 'A'
```

```
SELECT ID FROM USERS WHERE USERNAME = 'ADMIN' AND PASSWORD LIKE 'A%'
```

```
SELECT ID FROM USERS WHERE USERNAME = 'ADMIN' UNION TABLENAME FROM TABLES WHERE  
SUBSTR(TABLENAME,1,1) == 'A' '
```

2) work on optional question

3)a) type klist in terminal to get the output

b) only 1 ticket should be valid

c) find expiry time

d) two principals - myself and krbtgt/ANDREW.CMU.EDU

4) a) first four kerberos - after sorting with packet number

b) find the server name in c -> TGS (TGS- REQ) server name - AFS

- c) AS - REP for ticket in encrypted format, and TGS-REQ for ticket in encrypted format
- d) aes256-cts-hmac-sha1-96
- e) go to TGS REP to get TICKET_V