Network Security Term Project Report

maTLS : Implementation of the Middlebox Aware TLS Protocol

భారతీయ సాంకేతిక విజ్ఞాన సంస్థ హైదరాబాద్
भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Tejas Deshmukh (cs22mtech12005)

Aneesh Ajaroy Gantasala (cs19btech11010)

03.05.2023

PROJECT TITLE

maTLS : Implementation of the Middlebox Aware TLS Protocol

PROBLEM STATEMENT

Middleboxes are widely used for various purposes like Security enhancements, content filtering, etc. But, they are rendered useless with the introduction of the TLS protocol. Proposed solutions to include MBs were Encryption-based, TEE. But these have limited functionalities. Thus, we adopt TLS Extension Based solutions.

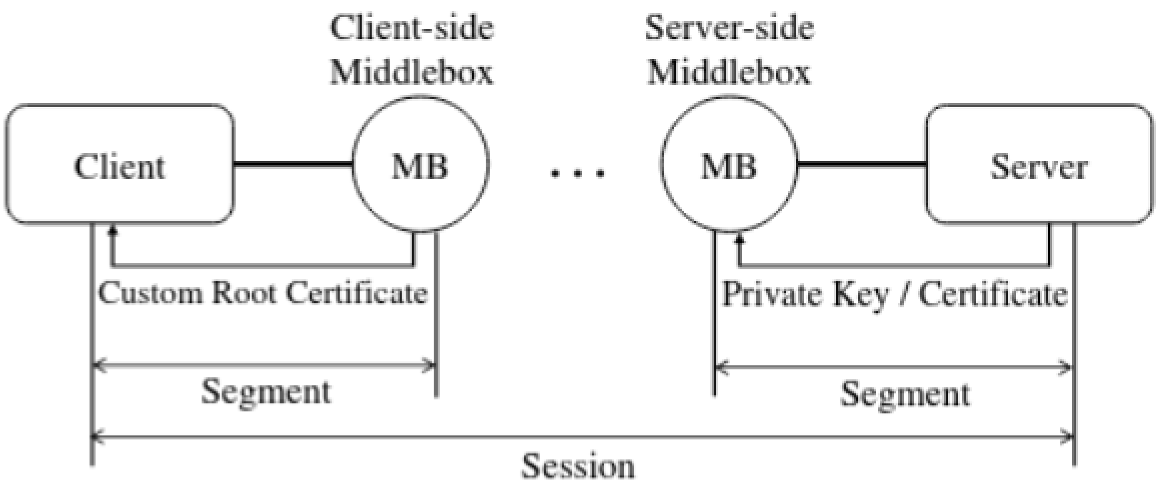
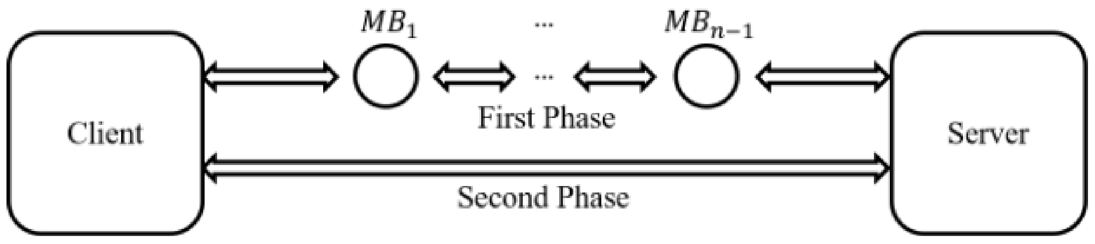


Fig1. Overview of SplitTLS[1]

Existing solutions like SplitTLS include the middleboxes in TLS sessions, but this compromises Confidentiality, Integrity and Authenticity. To achieve these goals, we implement a Middlebox Aware protocol (maTLS) to make the middleboxes visible to the client and make them publically auditable .

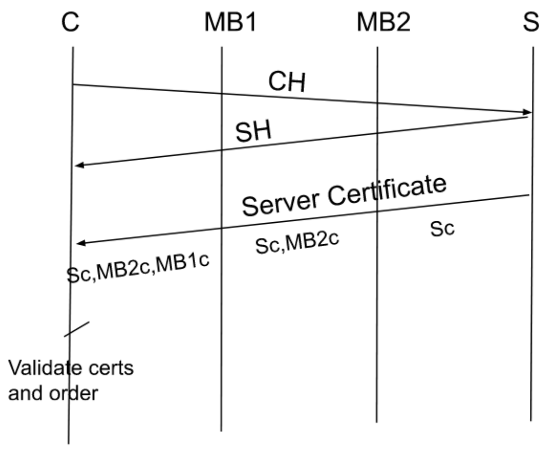
PROJECT DESCRIPTION



maTLS helps to achieve Authenticity, Confidentiality and Integrity. No need to install certificates in the trusted store of Client or for the Server to share its private key to the Middlebox which is a privacy and security threat, because we make the middleboxes themselves auditable. We go for a Bottom up maTLS design approach in which the Client and the Middlebox initiates TLS segments sequentially up to the Server. Two entities of each segment negotiate their security parameters individually. Security Goals to be achieved in the maTLS :

1) Authentication :

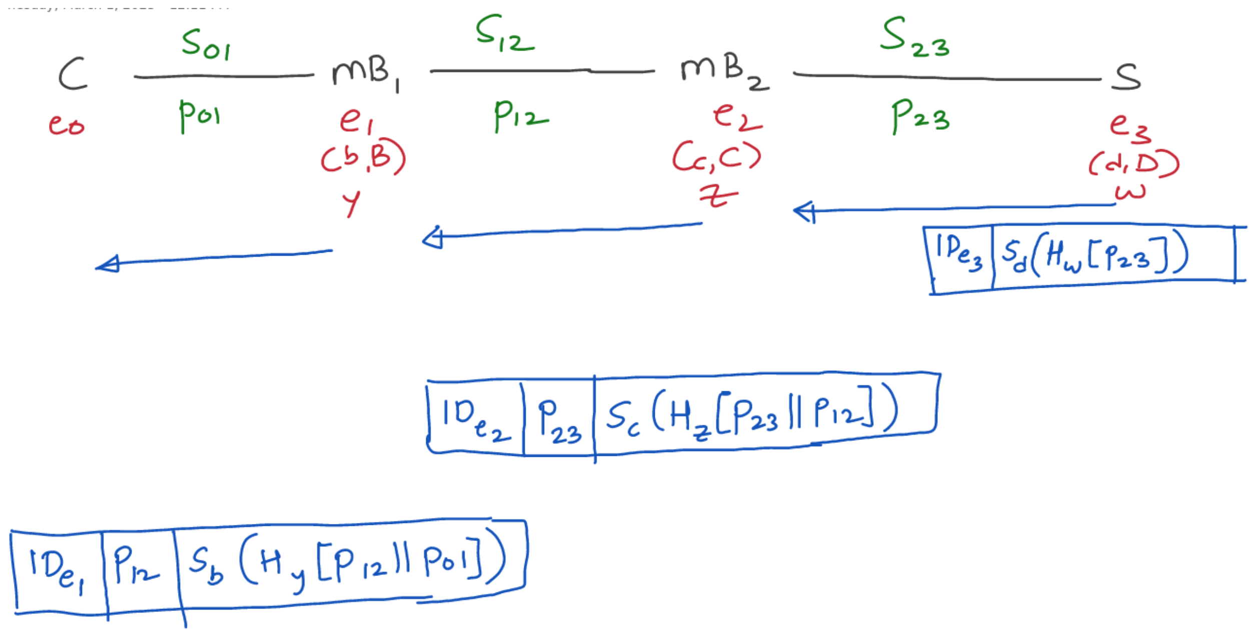
   Server and all the Middle Boxes were authenticated. This step is also called "EXPLICIT AUTHENTICATION". This helps with accountability. It is achieved by using certificates using PKI. In this step, the server sends its certificates to the Middlebox which inturn appends its own certificate and forwards it to the client. Thus, the client gets the certificate of all the middleboxes and the server. Now, the client can verify each of the certificates and authenticate Middleboxes and the server.



Explicit Authentication mechanism

2) Confidentiality :

Each maTLS segment has Segment Secrecy as well as Individual secrecy. Segment Secrecy ensures that each of the segments uses a higher TLS version and Strong Cipher suites. This ensures that the overall TLS connection that is constructed from these individual segments is secure, i.e using higher TLS version and Strong Cipher suites overall. Individual Secrecy on the other hand is about the uniqueness of the session key in each of these segments. Every segment has its own security association and each of them must have a unique session key, just in case if one of the keys is compromised, yet the other segments are still secure. If there is a MITM attacker sitting in between this uncompromised segment, he cannot decrypt the session. Thus, individual secrecy adds a layer of security and enhances confidentiality. Confidentiality is achieved by a method known as "SECURITY PARAMETER VERIFICATION". In this, the end points check the negotiated security parameters in between each segment.


C
S_{01}
P_{01}
e_0
mB_1
e_1
(b, B)
y
S_{12}
P_{12}
mB_2
e_2
(c, C)
z
S_{23}
P_{23}
S
e_3
(d, D)
w
ID_{e_3} | $S_d(H_w[P_{23}])$
ID_{e_2} | P_{23} | $S_c(H_z[P_{23} || P_{12}])$
ID_{e_1} | P_{12} | $S_b(H_y[P_{12} || P_{01}])$

Security Parameter Verification mechanism

3) Integrity :

If a middlebox tries to modify the data sent in to and fro the client and server, the client shall be notified about it. This ensures that the integrity of the system is maintained. This includes doing the Data Source Authentication, Modification Accountability, and maintaining the Path Integrity. In Data Source Authentication, Client confirms that the message that is received is from the Server itself, and not from anyone else. In Modification Accountability, Client comes to know of the Middleboxes that have done any modifications to the data exchanged and in Path Integrity, Client can verify that the message has passed through the authenticated Middleboxes in the established order and there is no other entity in between. Integrity is achieved by doing "VALID MODIFICATION CHECKS".

In our project, we are successful in carrying out the Explicit Authentication and the secure sharing of the Security Parameters for the security parameters verification. Thus, able to achieve the Authentication and securely shared the parameters for carrying out the Confidentiality.

One of the most important challenges faced was to carry out the security parameter verification. Client receives the Security Parameter Blocks of the middlebox and the server which contains the ID, Security parameters and the Signature. Understanding the complex unforgeable security block structure, and to extract its components and validate them in a proper order.

Identifier	Sec Para of segment in the direction of server	Signature with ei's private key on { Hash(done by ak of ei with client) on [Sec paras in both directions]}

Fig 2. Security Parameter Block Generic

$$ID_i||p_{i,i+1}||Sign(sk_i, Hmac(ak_{i,0}, p_{i-1,i}||p_{i,i+1}))$$

Fig 3. SPB of MiddleBox

$$ID_n||Sign(sk_n, Hmac(ak_{n,0}, p_{n-1,n}))$$

Fig 4. SPB of Server

Another challenge was to understand the Modification log block and its use for integrity.

ID	Hash of previous message	HMAC with ak on (Hash of received msg + Hash of sent msg)	Previous ML

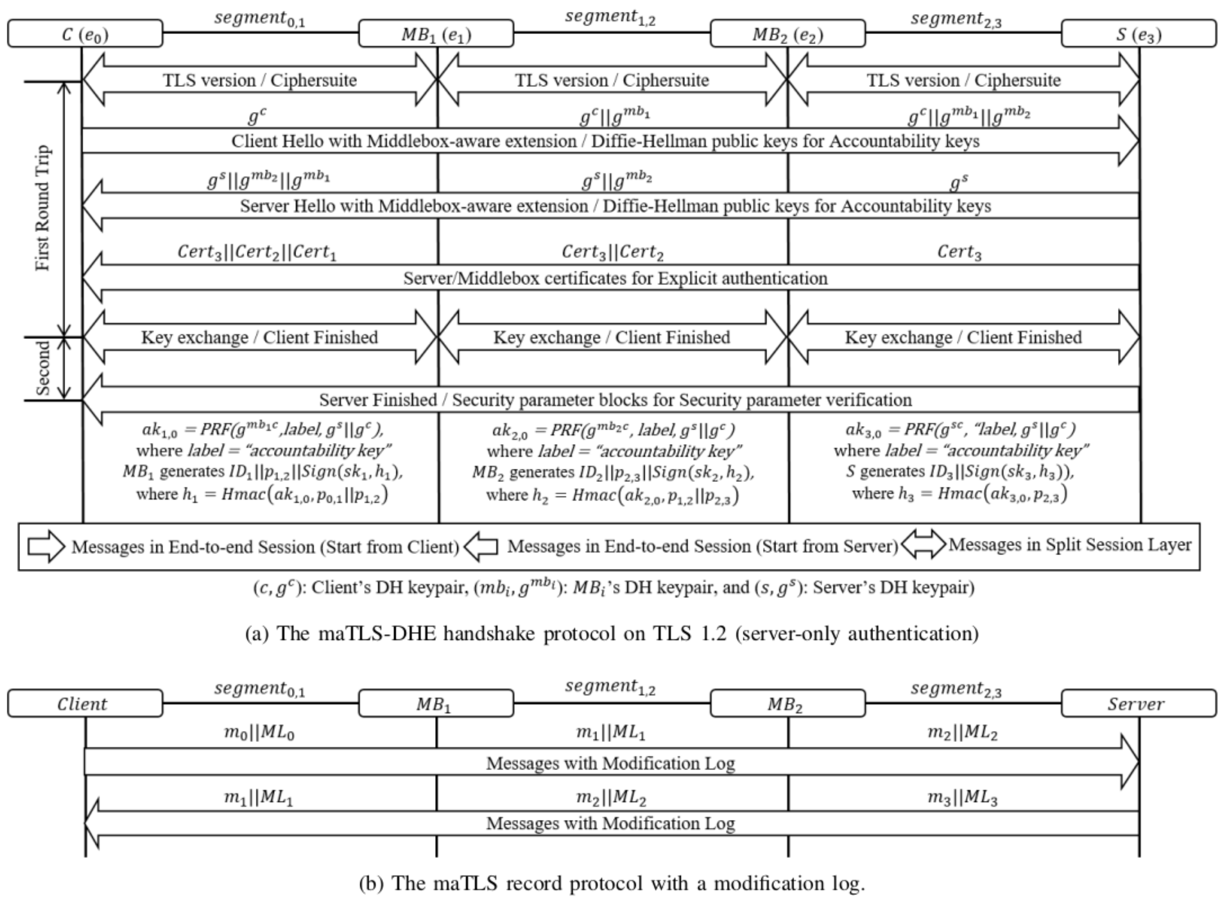
Fig 5. Modification Log Block Generic

$$ID_i||H(m_{i+1})||Hmac(ak_{i,0}, H(m_i)||H(m_{i+1}))||ML_{i+1}$$

Fig 6. MLB of Middlebox

$$ID_n||Hmac(ak_{n,0}, H(m_n))$$

Fig 7. MLB of Server


$C\ (e_0)$
$segment_{0,1}$
$MB_1\ (e_1)$
$segment_{1,2}$
$MB_2\ (e_2)$
$segment_{2,3}$
$S\ (e_3)$
TLS version / Ciphersuite
First Round Trip
Second
g^c
$g^c||g^{mb_1}$
$g^c||g^{mb_1}||g^{mb_2}$
Client Hello with Middlebox-aware extension / Diffie-Hellman public keys for Accountability keys
$g^s||g^{mb_2}||g^{mb_1}$
$g^s||g^{mb_2}$
g^s
Server Hello with Middlebox-aware extension / Diffie-Hellman public keys for Accountability keys
$Cert_3||Cert_2||Cert_1$
$Cert_3||Cert_2$
$Cert_3$
Server/Middlebox certificates for Explicit authentication
Key exchange / Client Finished
Server Finished / Security parameter blocks for Security parameter verification
$ak_{1,0} = PRF(g^{mb_1c}, label, g^s||g^c)$, where $label$ = "accountability key" MB_1 generates $ID_1||p_{1,2}||Sign(sk_1, h_1)$, where $h_1 = Hmac(ak_{1,0}, p_{0,1}||p_{1,2})$
$ak_{2,0} = PRF(g^{mb_2c}, label, g^s||g^c)$ where $label$ = "accountability key" MB_2 generates $ID_2||p_{2,3}||Sign(sk_2, h_2)$, where $h_2 = Hmac(ak_{2,0}, p_{1,2}||p_{2,3})$
$ak_{3,0} = PRF(g^{sc}$, "label, $g^s||g^c)$ where $label$ = "accountability key" S generates $ID_3||Sign(sk_3, h_3))$, where $h_3 = Hmac(ak_{3,0}, p_{2,3})$
Messages in End-to-end Session (Start from Client)
Messages in End-to-end Session (Start from Server)
Messages in Split Session Layer
(c, g^c): Client's DH keypair, (mb_i, g^{mb_i}): MB_i's DH keypair, and (s, g^s): Server's DH keypair)
(a) The maTLS-DHE handshake protocol on TLS 1.2 (server-only authentication)
Client
MB_1
MB_2
Server
$m_0||ML_0$
$m_1||ML_1$
$m_2||ML_2$
Messages with Modification Log
$m_1||ML_1$
$m_2||ML_2$
$m_3||ML_3$
(b) The maTLS record protocol with a modification log.

Flow diagram of overall maTLS working

COMPLETED POC IMPLEMENTATION

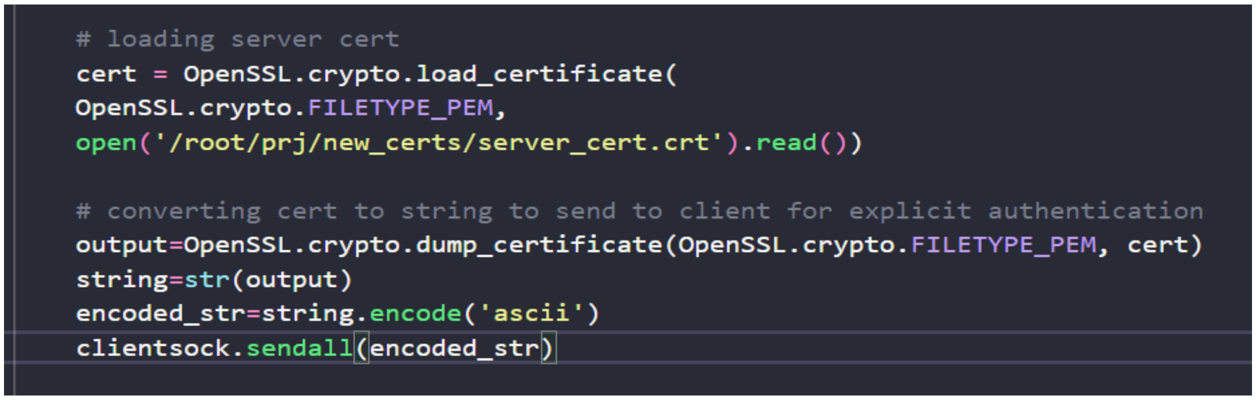
The steps in our project include :

Creation of certificates for the Client, Middlebox and the Server.

Tried to understand and change the openssl library. But, it was too complicated and time consuming as a change in one function doesn't guarantee that the TLS handshake will succeed. We tried to understand the handshake code and put in some custom extensions in the Client Hello message to share the Diffie Hellman keys that will be needed for the generation of the accountability keys, but that did not succeed. So, for this we assume that the accountability keys are inherently present, i.e by some mechanism previous to the TLS handshake, these passphrases or keys were exchanged with the server and the middlebox. These keys act as accountability keys that will be used for the keyed hashing. Each entity has an accountability symmetric key held by that entity with the endpoints server and the client.

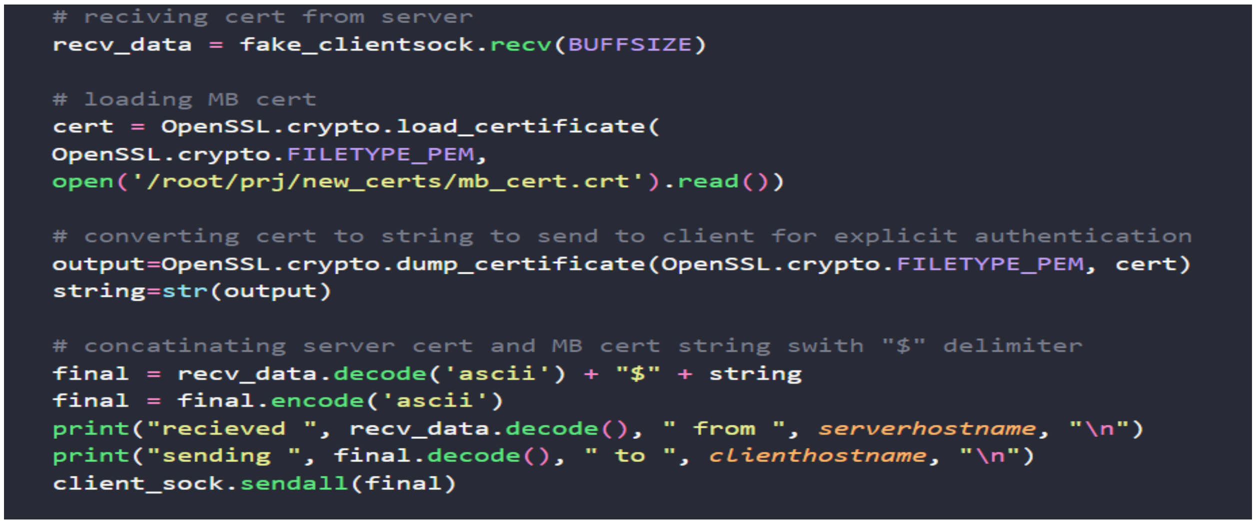
We also tried appending the certificates in the Server Hello message for the explicit authentication. But, there was no appropriate function found in openssl library to append these certificates in the SH message. Thus, we went forward to using the pyopenssl. In this again, instead of appending the certificates in the SH message, we are sending them prior to the TLS handshake. So, Server sends a message which has its certificate. Middlebox receives it and appends its own certificate to that message and passes that message to the client. The Client then goes forward for verification of these certificates. It separates out the certificates and stores them as .crt. Then it opens them and by using the public key of the Root CA, it verifies the certificates of the Middlebox and that of the Server. Thus, in this way, we were able to achieve the EXPLICIT AUTHENTICATION.

Code Snippets :



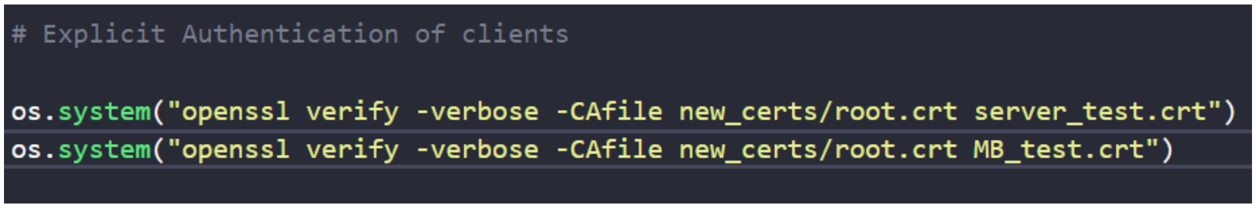
```
# loading server cert
cert = OpenSSL.crypto.load_certificate(
OpenSSL.crypto.FILETYPE_PEM,
open('/root/prj/new_certs/server_cert.crt').read())

# converting cert to string to send to client for explicit authentication
output=OpenSSL.crypto.dump_certificate(OpenSSL.crypto.FILETYPE_PEM, cert)
string=str(output)
encoded_str=string.encode('ascii')
clientsock.sendall(encoded_str)
```

Server certificate sending



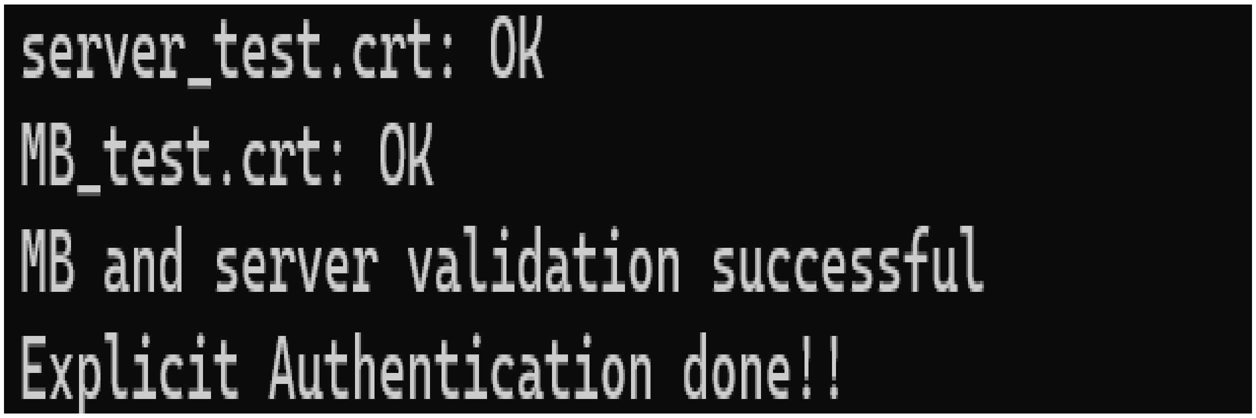
```
# reciving cert from server
recv_data = fake_clientsock.recv(BUFFSIZE)

# loading MB cert
cert = OpenSSL.crypto.load_certificate(
OpenSSL.crypto.FILETYPE_PEM,
open('/root/prj/new_certs/mb_cert.crt').read())

# converting cert to string to send to client for explicit authentication
output=OpenSSL.crypto.dump_certificate(OpenSSL.crypto.FILETYPE_PEM, cert)
string=str(output)

# concatinating server cert and MB cert string swith "$" delimiter
final = recv_data.decode('ascii') + "$" + string
final = final.encode('ascii')
print("recieved ", recv_data.decode(), " from ", serverhostname, "\n")
print("sending ", final.decode(), " to ", clienthostname, "\n")
client_sock.sendall(final)
```

MB appending and Sending



```
# Explicit Authentication of clients

os.system("openssl verify -verbose -CAfile new_certs/root.crt server_test.crt")
os.system("openssl verify -verbose -CAfile new_certs/root.crt MB_test.crt")
```

Explicit Authentication in Client



```
server_test.crt: OK
MB_test.crt: OK
MB and server validation successful
Explicit Authentication done!!
```

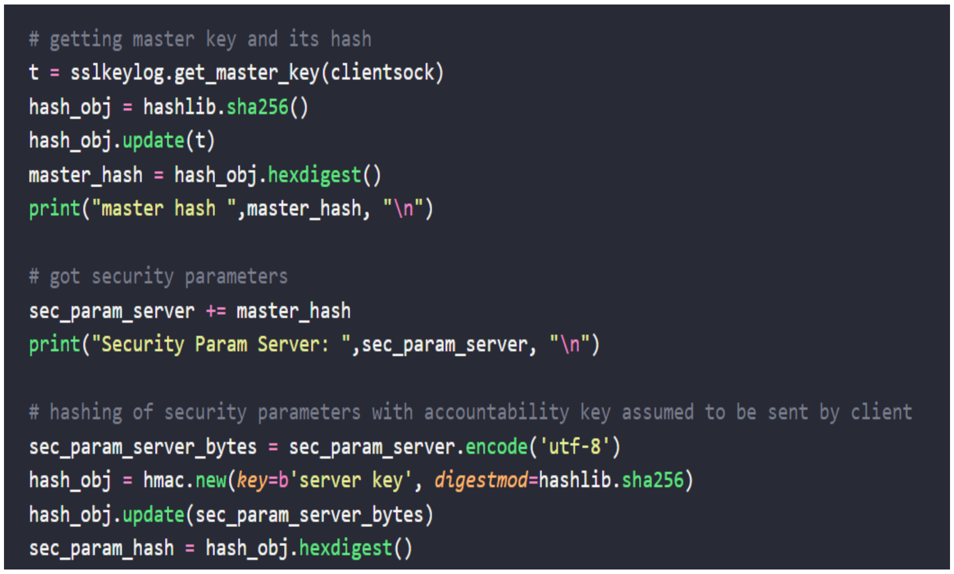
Output of Explicit Authentication

In the next step, we do the TLS handshake with the neighboring entities. Next step was to get the security parameters ready. A security parameter consists of the following things :

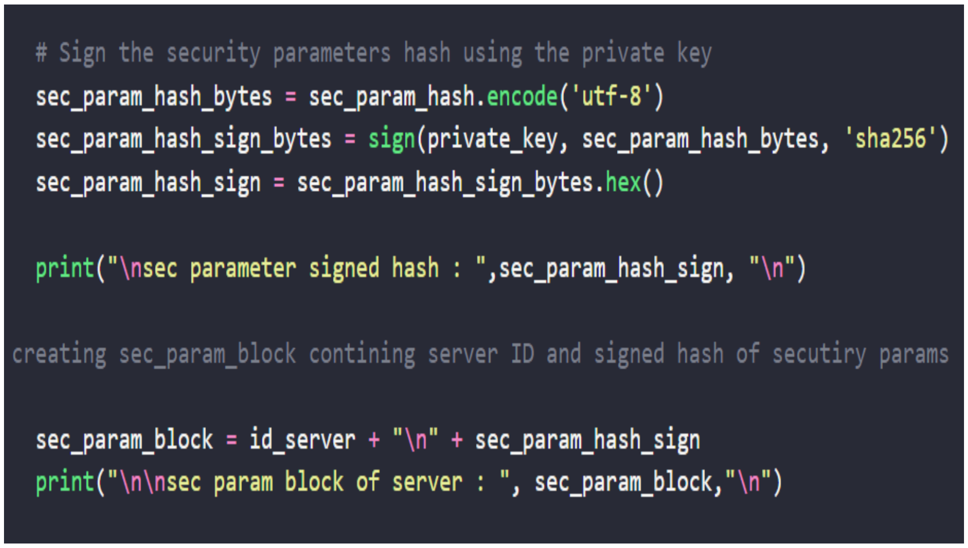
[Chosen TLS ver, Negotiated CC, H(MS), Hased Transcript of TLS HS=verify_data in FIN]

We were able to extract the first 3 components, namely the Chosen TLS version, the negotiation Cipher Suite and the Master secret that were negotiated in the above TLS handshake phase. We computed the hash of this master secret by using the “hashlib” library. We use the “logkeyfile” to get the master secret. But, no appropriate function was found that could extract the Transcript of the done TLS HS. Thus, we are not including it in the Security parameters. Then, we go on to the creation of the security parameter block. In this, we took the Identifier of the entity as the hash of its public key. We append the ID, the security parameters, and the signed hash of the security parameters in both the directions. The signature was done by the private key of that particular entity. Then, we pass these security parameters from the server to the middlebox. The middlebox forwards this along with its own SPB and sends it to the client for the verification. The client loads the public keys of both the entities and verifies these SPBs. Thus, security parameter block verification is achieved in this way.

Code Snippets :

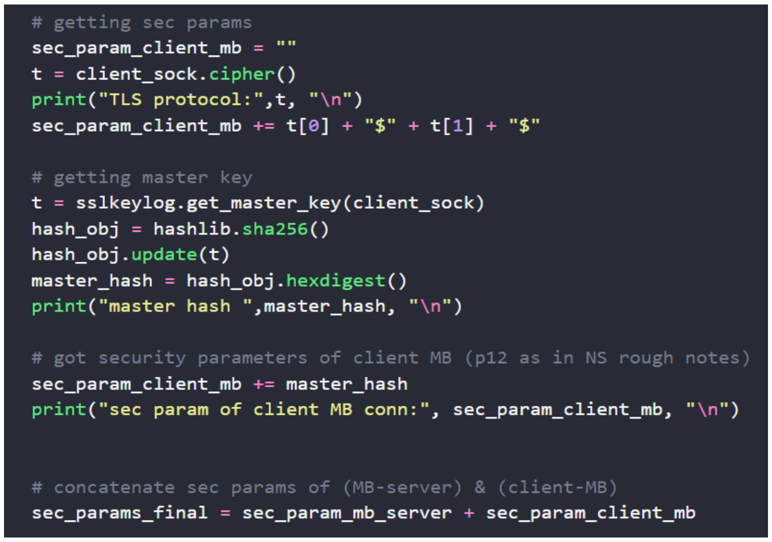


```python
# getting master key and its hash
t = sslkeylog.get_master_key(clientsock)
hash_obj = hashlib.sha256()
hash_obj.update(t)
master_hash = hash_obj.hexdigest()
print("master hash ",master_hash, "\n")

# got security parameters
sec_param_server += master_hash
print("Security Param Server: ",sec_param_server, "\n")

# hashing of security parameters with accountability key assumed to be sent by client
sec_param_server_bytes = sec_param_server.encode('utf-8')
hash_obj = hmac.new(key=b'server key', digestmod=hashlib.sha256)
hash_obj.update(sec_param_server_bytes)
sec_param_hash = hash_obj.hexdigest()
```

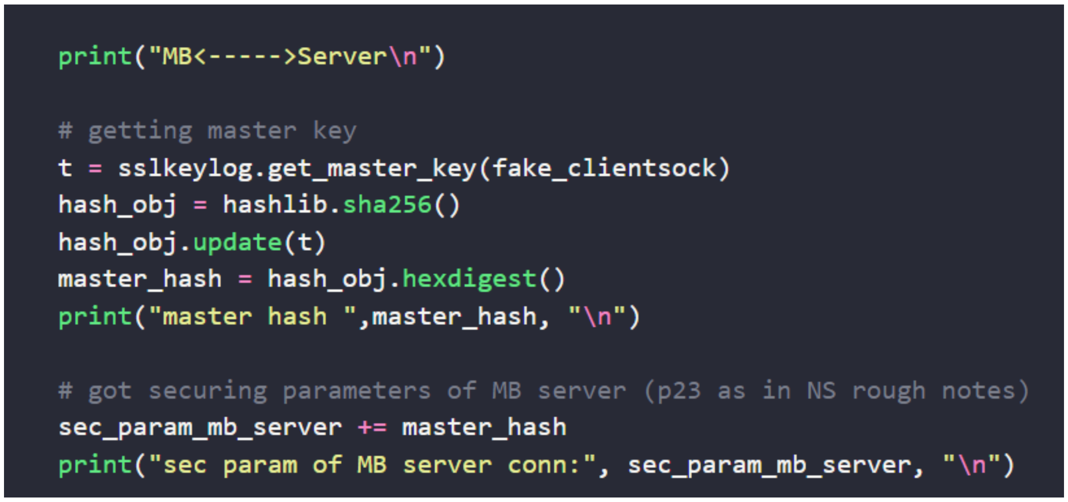


```
# Sign the security parameters hash using the private key
sec_param_hash_bytes = sec_param_hash.encode('utf-8')
sec_param_hash_sign_bytes = sign(private_key, sec_param_hash_bytes, 'sha256')
sec_param_hash_sign = sec_param_hash_sign_bytes.hex()

print("\nsec parameter signed hash : ",sec_param_hash_sign, "\n")

creating sec_param_block contining server ID and signed hash of secutiry params

sec_param_block = id_server + "\n" + sec_param_hash_sign
print("\n\nsec param block of server : ", sec_param_block,"\n")
```

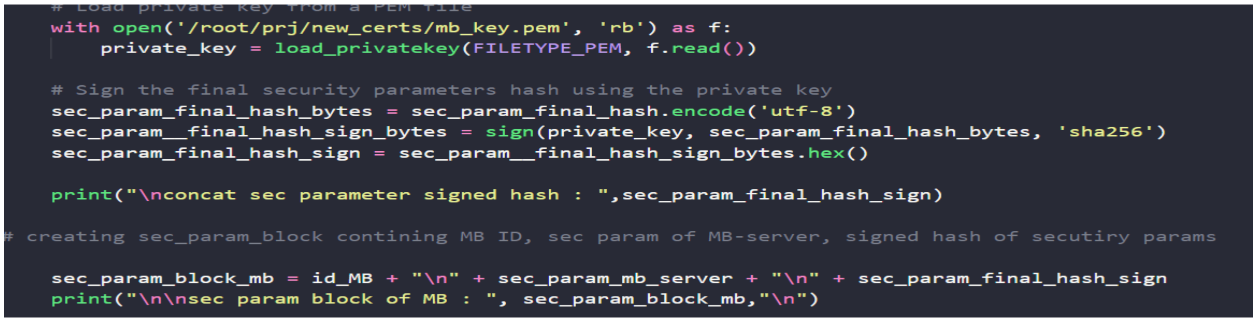
Creating Security Parameter Block of Server



```
# getting sec params
sec_param_client_mb = ""
t = client_sock.cipher()
print("TLS protocol:",t, "\n")
sec_param_client_mb += t[0] + "$" + t[1] + "$"

# getting master key
t = sslkeylog.get_master_key(client_sock)
hash_obj = hashlib.sha256()
hash_obj.update(t)
master_hash = hash_obj.hexdigest()
print("master hash ",master_hash, "\n")

# got security parameters of client MB (p12 as in NS rough notes)
sec_param_client_mb += master_hash
print("sec param of client MB conn:", sec_param_client_mb, "\n")


# concatenate sec params of (MB-server) & (client-MB)
sec_params_final = sec_param_mb_server + sec_param_client_mb
```

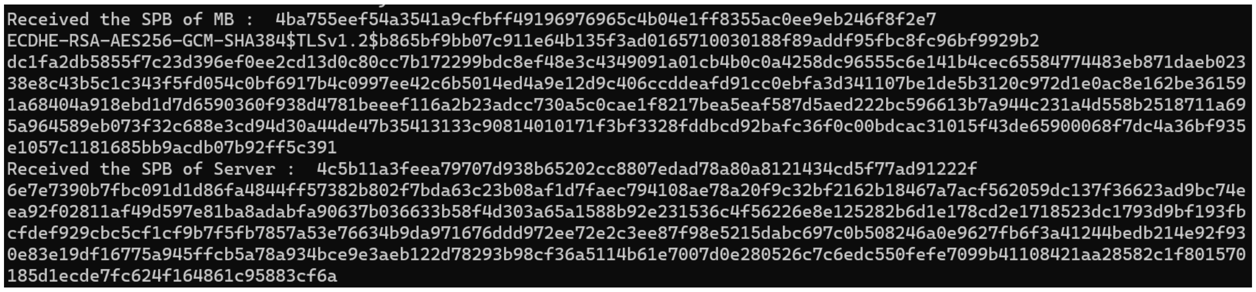


```python
print("MB<----->Server\n")

# getting master key
t = sslkeylog.get_master_key(fake_clientsock)
hash_obj = hashlib.sha256()
hash_obj.update(t)
master_hash = hash_obj.hexdigest()
print("master hash ",master_hash, "\n")

# got securing parameters of MB server (p23 as in NS rough notes)
sec_param_mb_server += master_hash
print("sec param of MB server conn:", sec_param_mb_server, "\n")
```



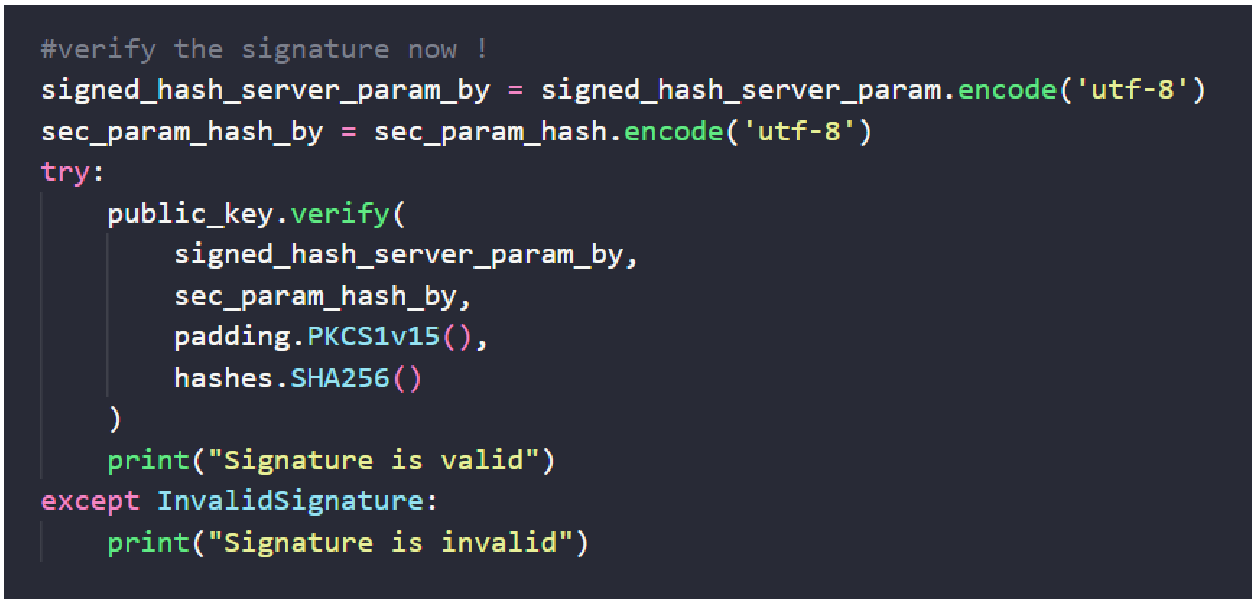
```python
    # Load private key from a PEM file
    with open('/root/prj/new_certs/mb_key.pem', 'rb') as f:
        private_key = load_privatekey(FILETYPE_PEM, f.read())

    # Sign the final security parameters hash using the private key
    sec_param_final_hash_bytes = sec_param_final_hash.encode('utf-8')
    sec_param__final_hash_sign_bytes = sign(private_key, sec_param_final_hash_bytes, 'sha256')
    sec_param_final_hash_sign = sec_param__final_hash_sign_bytes.hex()

    print("\nconcat sec parameter signed hash : ",sec_param_final_hash_sign)

# creating sec_param_block contining MB ID, sec param of MB-server, signed hash of secutiry params

    sec_param_block_mb = id_MB + "\n" + sec_param_mb_server + "\n" + sec_param_final_hash_sign
    print("\n\nsec param block of MB : ", sec_param_block_mb,"\n")
```

Security Parameter Block of the Middlebox



```
Received the SPB of MB :  4ba755eef54a3541a9cfbff49196976965c4b04e1ff8355ac0ee9eb246f8f2e7
ECDHE-RSA-AES256-GCM-SHA384$TLSv1.2$b865bf9bb07c911e64b135f3ad0165710030188f89addf95fbc8fc96bf9929b2
dc1fa2db5855f7c23d396ef0ee2cd13d0c80cc7b172299bdc8ef48e3c4349091a01cb4b0c0a4258dc96555c6e141b4cec65584774483eb871daeb023
38e8c43b5c1c343f5fd054c0bf6917b4c0997ee42c6b5014ed4a9e12d9c406ccddeafd91cc0ebfa3d341107be1de5b3120c972d1e0ac8e162be36159
1a68404a918ebd1d7d6590360f938d4781beeef116a2b23adcc730a5c0cae1f8217bea5eaf587d5aed222bc596613b7a944c231a4d558b2518711a69
5a964589eb073f32c688e3cd94d30a44de47b35413133c90814010171f3bf3328fddbcd92bafc36f0c00bdcac31015f43de65900068f7dc4a36bf935
e1057c1181685bb9acdb07b92ff5c391
Received the SPB of Server :  4c5b11a3feea79707d938b65202cc8807edad78a80a8121434cd5f77ad91222f
6e7e7390b7fbc091d1d86fa4844ff57382b802f7bda63c23b08af1d7faec794108ae78a20f9c32bf2162b18467a7acf562059dc137f36623ad9bc74e
ea92f02811af49d597e81ba8adabfa90637b036633b58f4d303a65a1588b92e231536c4f56226e8e125282b6d1e178cd2e1718523dc1793d9bf193fb
cfdef929cbc5cf1cf9b7f5fb7857a53e76634b9da971676ddd972ee72e2c3ee87f98e5215dabc697c0b508246a0e9627fb6f3a41244bedb214e92f93
0e83e19df16775a945ffcb5a78a934bce9e3aeb122d78293b98cf36a5114b61e7007d0e280526c7c6edc550fefe7099b41108421aa28582c1f801570
185d1ecde7fc624f164861c95883cf6a
```

Received Security Parameter Blocks



```
#verify the signature now !
signed_hash_server_param_by = signed_hash_server_param.encode('utf-8')
sec_param_hash_by = sec_param_hash.encode('utf-8')
try:
    public_key.verify(
        signed_hash_server_param_by,
        sec_param_hash_by,
        padding.PKCS1v15(),
        hashes.SHA256()
    )
    print("Signature is valid")
except InvalidSignature:
    print("Signature is invalid")
```

Verification of security parameter blocks

EXPERIMENTAL SETUP

Used the container host provided in the Secure Chat assignment. It consists of three containers : Alice, Trudy and Bob. We use the Alice1 container as the Client, the Trudy1 container as Middlebox, and the Bob1 container as the server. Then we poison the DNS cache to simulate that the Middlebox is in between without the Client knowing it. Client is sending its first message as if it wants to connect to the server. We write the “client.py”, “middlebox.py” and “server.py” in the respective containers.

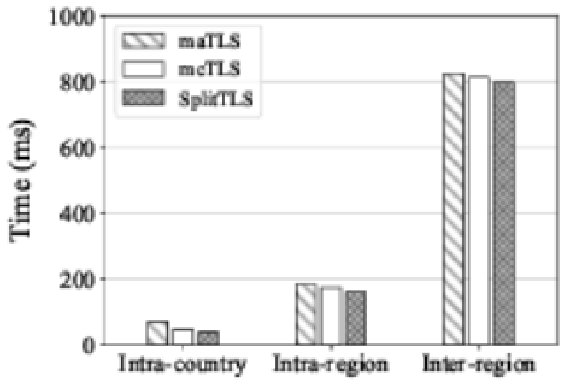
In “client.py”, we load the certificate of the client, get it verified. Then we establish the TCP connection. We receive the 2 certificates in an appended way from the middlebox. These certificates are separated out and stored separately. Then, we validate these certificates. This ensures that the EXPLICIT AUTHENTICATION phase has been completed. After that, we establish a TLS connection. Once the Handshake is completed, we extract the security parameters of the Client with the middlebox. These will be used for validations. After the handshake, we receive the security parameter blocks of the middlebox and the server from the middlebox. Then, we move on to validating these security parameter blocks.

In “middlebox.py”, initial things are the same as in the client.py. Then we establish 2 tcp connections, one with the client, and the other with the server. We get the certificate from the server, append the middlebox certificate to this message and pass it onto the client. Then, TLS connections are established and the middlebox then creates two security parameters, one with the client and the other with the server. Then, the security parameter block of the middlebox is created. After receiving the SPB from the server, middlebox also sends its own SPB to the client for the security parameter verification phase.

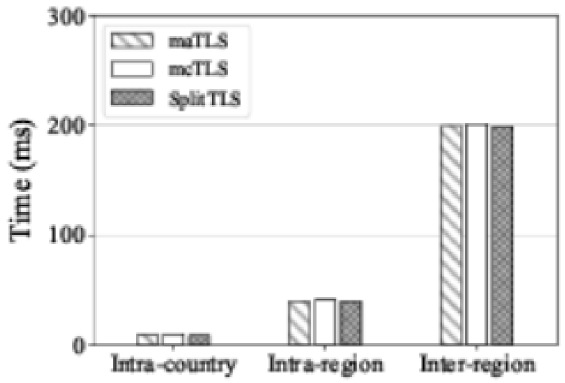
In “server.py”, initial things are the same as in client.py. We establish a TCP connection. Then, the server sends its own certificate to the middlebox which then appends its own and sends it to the client. Then, a TLS connection is established between the server and the middlebox. It extracts the security parameters from this TLS connection and makes the Security parameter block. It then passes this SPB to the middlebox, which also passes its own SPB along with this to the client for the security parameter verification phase.

PERFORMANCE METRICS

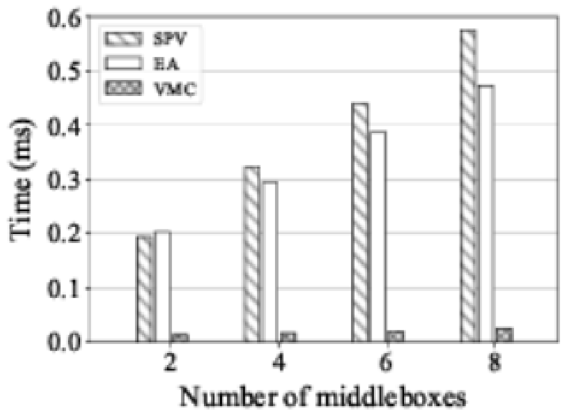
The performance metrics as given in the base paper are as follows :



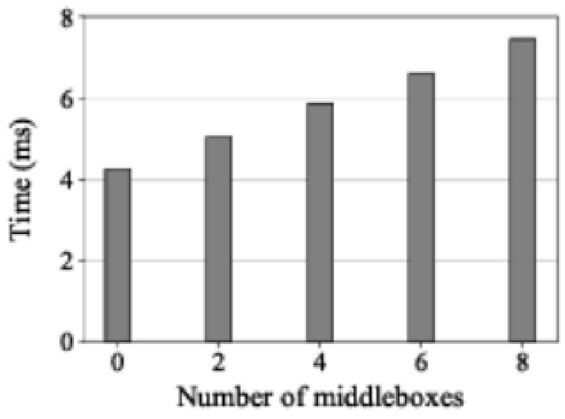
(a) HTTP Load Time



(b) Data Transfer Time



(c) Integrity Verification Time



(d) CPU Processing Time

CONCLUSIONS & FUTURE SCOPE

maTLS protocol implementation has three main parts to be implemented :

1) EXPLICIT AUTHENTICATION : To verify the authenticity of the server as well as the middleboxes.
2) SECURITY PARAMETER VERIFICATION : To verify the segment and individual secrecy.
3) MODIFICATION LOG BLOCKS : To achieve integrity checks on the data to be sent which includes data source authentication as well as the path integrity check.

We were able to successfully achieve EXPLICIT AUTHENTICATION. We are also able to successfully establish TLS segments and get the security parameters to create the Security Parameter Blocks. These blocks are sent to the client and our last progress is to try to verify these security parameter blocks in order to ensure confidentiality.

Future scope holds the implementation of the MODIFICATION LOG BLOCKS.

Our thoughts on maTLS protocol :

A protocol like maTLS might not really be needed as even if it is able to achieve the goals, there is still a privacy issue. Middlebox can still decrypt the traffic, and if it gets compromised, could leak severe information. Instead of this, we think that Machine learning techniques could be applied to do the job on the encrypted traffic itself. Lets say, we wish to classify the network traffic, train a ML model over the flow features like Inter Arrival Time, direction(forward/reverse bytes and packets) and the packet length. The use of a right ML algorithm(here supervised-Logistic or the new booming self-supervised) can give an accuracy and a recall of more than 98%.

REFERENCES

1) maTLS: How to Make TLS middlebox-aware? - NDSS Symposium 2022, https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01B-6_Lee_paper.pdf
2) D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. L ́opez, K. Papagiannaki, P. R. Rodriguez, and P. Steenkiste, “Multi-context tls (mctls): Enabling secure in-network functionality in tls,” in ACM SIGCOMM Computer Communication Review, vol. 45, no. 4. ACM, 2015, pp. 199–212.

CONTRIBUTIONS

Tejas :

- Understood the working of maTLS protocol and had a meeting with Aneesh.
- Generated the certificates
- Wrote the codes for client and helped in coding of middlebox.

- Created the security parameters and the security parameter blocks.
- Tried to do the security parameter verification.

Aneesh :

- Understood the working of maTLS protocol and had a meet with Tejas
- Tried the openssl library changes. Finally came up with the conclusion of using pyopenssl.
- Wrote the codes for server and helped in coding of middlebox.
- Made appropriate changes in the codes to implement and pass the Security parameter blocks.
- Tried to do the security parameter verification.

ANTI-PLAGIARISM STATEMENT

ANTI PLAGIARISM STATEMENT <Include it in your report. This statement has been revised as you are allowed to use any publicly available tools/repos/scripts, including ChaptGPT's help for capturing the flags in this assignment>

We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, ChatGPT tips, packages, datasets, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment/project in any other course lab, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarized the work of other students in this group. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand my responsibility to report honor violations by other students if we become aware of it.

Names:

Tejas Deshmukh (cs22mtech12005)

Aneesh Ajaroy Gantasala (cs19btech11010)

Date: 03 May 2023

Signature: Tejas, Aneesh