Best Current Practise

OCSBC – UCaaS security aspects

Category: Informational

February 2024, Version 1.00

Revision History

Version	Author	Description of Changes	Date Revision Completed
0.00	Matej Maric	Initial version	
1.00	Matej Maric	Atcpd debug logs captured, generic TLS intro	09_07_2024

Abstract

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

The configurations provided in this document SHOULD NOT be treated as RECOMMENDED. The information is intended to provide guidance as to the OCSBC behaviour when configurations listed in this document are applied.

This document is intended to provide the reader with information regarding configuration of an OCSBC to provide user authentication via several RADIUS servers.

Applicability

The details provided are relevant to physical & virtual Oracle Communications Session Border Controller (OCSBC) instances.

Table of content

Contents

Network function

Focus of this BCP is SBC that coexist as part of UCaaS demo LAB that terminates SIP TLS connections towards Microsoft (Direct Routing), Webex (Calling), Zoom (Phone) and Google (SipLink). SBC acts as well as media (RTP) termination point interworking in such deployments SRTP from internet legs into core RTP legs. As a best practice in general, security wise, we'll be checking lab's OCOM. Calling devices here are UCaaS native clients and lab's IMS registered softphones simulating PSTN.

TLS generic introduction

TLS 1.2 – This document will not revert to older TLS versions as are deprecated today and should not be used at all.

In TLS 1.2 RSA, DH and DHE cipher suites are available.

Key exchange algh being RSA, handshake on high level looks as depicted below:

client hello - (client exposes TLS version, generates random, exposes cipher suits supported)

server hello - (server agrees on TLS version(or not), sends it's 'random', and picks one of the cipher suits - picks in this case RSA one)

certificates(sent by server) - server sends its certificate chain

"client key exchange", "encrypted handshake"(sent by client) - before sending this messages client authenticates the server identity by checking the server-side CA public certificate chain against its trusted store. If the check is done successfully the client proceeds with "client key exchange". In this message client creates a pre-master secret and encrypts it with learned server public key(server's end-entity certificate, nothing to do with CA public certificates)

upon receipt of "client key exchange" server should be able to decrypt it with its private key as there are mathematical relations between its private and public key. That's the point server should learn same pre-master that client generated

Finally, both sides create a session key as SESSION_KEY= HASH of(premaster secret, client random, server random) and that key is used as an encryption/decryption key for traffic as of that point on

RSA cipher suits are with obvious downside:

Note above that "client random" and "server random" are per session values but they are exchanged in clear text! Once security is compromised and one gets the server private key then the attacker has a clear view over all historically saved sessions. This is due to the fact that server private-key exists as variable in session key calculation while other variables in calculation are exchanged in clear text.

As requirements on security evolved we've got new DH, DHE cipher suits and main idea here was to rule out server's private key from session-key calculation.

So still staying in TLSv1.2 but with DH and DHE cipher suits in use that handshake would look as follows:

client hello - (client exposes TLS version, generates random, exposes cipher suits supported)

server hello - (server agrees on TLS version(or not), sends it's 'random', and picks one of the cipher suits - picks in this case DHE one)

certificates(sent by server) - server sends its certificate chain

"server key exchange"(sent by server) - this is first message that differs compared with RSA cipher suite in use. Here server sends its public key. Which public key? This key is part of the key-exchange process and has nothing to do with either server's public end entity certificates nor with CA public certs. This public key relates to DH algorithm that is pure math on how both sides may come to the same session key without involving server's private key into the picture(will explain later low level). Also, server puts a digital signature over this message with its private key(note there is nothing to decrypt here on client side, just for the client to check the signature given it learned server's certificate chain)

"client key exchange" - as same with RSA client will verify server's chain of trust getting its certificates, also, it will store the servers public key based on servers public key it will create its own public key and send to the server. saying again here, this public key has nothing tto do with any certificate and is part of DH key exchange process. Client sends this public key to the server

At this point both client and server have enough material to come to the same pre-master key that will be used for session key calculation. and SESSION KEY == HASH (premaster-secret, client random, server random)

Now, please note that as long as the final symmetric session encryption/decryption key seems to have the same formula there are big differences. So let's uncover some facts here:

premaster secret is independent of server's private key in new calculation

with pure DH, public keys in server-key-exchange and client-key-exchange remain the same per session that leads us potentially to the same threat as TLS had with RSA key-exchange principle. Having a piece of static info from client and server one might decrypt all historical sessions. This is the reason pure DH cipher suites do not exist in TLS1.3

with DHE(E stands for ephemeral) there are new public key's created per session on client and server side client. So this is were we should be in 2024. Compromising private key with DHE is not an issue, compromising private piece of info on client/server side that accounts in public key creation may affect only a single TLS session but not the whole communication history!

As the next logical question is what kind of public keys I'm talking about in DH(E) as part of key exchange process - I'll try to illustrate with a simple math. But let's go a bit lower into Materia. With TLS DH(E) cipher suite both client and server will create private&public key pair(again, nothing to do with certificates) and this looks in numbers like this.

server creates its private key a=5(called prime), defines a low number g=3(public piece of info) and defines modulo number p=7 (public piece of info)

server calculates its public key as A=g^a MOD p == 3^5 MOD 7 == 5

server sends to client the following: A,p,g

client creates its own private key b=4 and calculates its public key as B=g^b MOD p == 81 MOD 7 ==4

client sends its public key B=7 to the server

at this point with some math both sides should calculate the same pre-master key!

Server calculation for pre-master key s=B^a MOD 7 == 1024 MOD 7 == 2

Client calculation for pre-master key s=A^b MOD 7 == 625 MOD 7 == 2

math behind is (g^a)^b MOD p = (g^b)^a MOD p = g^a^b MOD p

"s" here stands for pre-master secret and later along with client random and server random builds session encryption/decryption key

In DHE a and b(as private keys) change for each TLS session and compromising one pair of keys may uncover only one TLS session.

Moving now to TLSv1.3. RSA and Static DH ciphers are ruled out, only DHE and ECDHE(variation of DHE whereas client and server private keys sit on an elliptic curve, and you don't want to see math for that - principle for key-exchange is same in nutshell as for DHE) are present. List of cipher suits reduced in TLSv1.3 to only 5 compared to 37 supported in TLSv1.2 and handshake looks as:

Client hello - looks same as with DHE in TLSv1.2 apart that client assumes key exchange algorithm that server will pick and sends its public key (DHE materials) straight away.

Server hello - looks the same as in TLSv1.2 DHE apart that message contains here also server certificates and server "finished message". Moreover, certificates and server finished message are sent encrypted as server has all details already to calculate the session key

Upon receipt of server hello client will authenticate the server and generate the session keys based on received server's public key (same math as in TLSv1.2 DHE)

In summary, with TLSv1.3 every piece of information after Client/Server Hello exchange is encrypted with future session-key. Key exchange in TLSv1.2 came into picture only after successful client-server authentication and in TLSv1.3 both authentication and session keys are established in the first two handshake messages.

Software

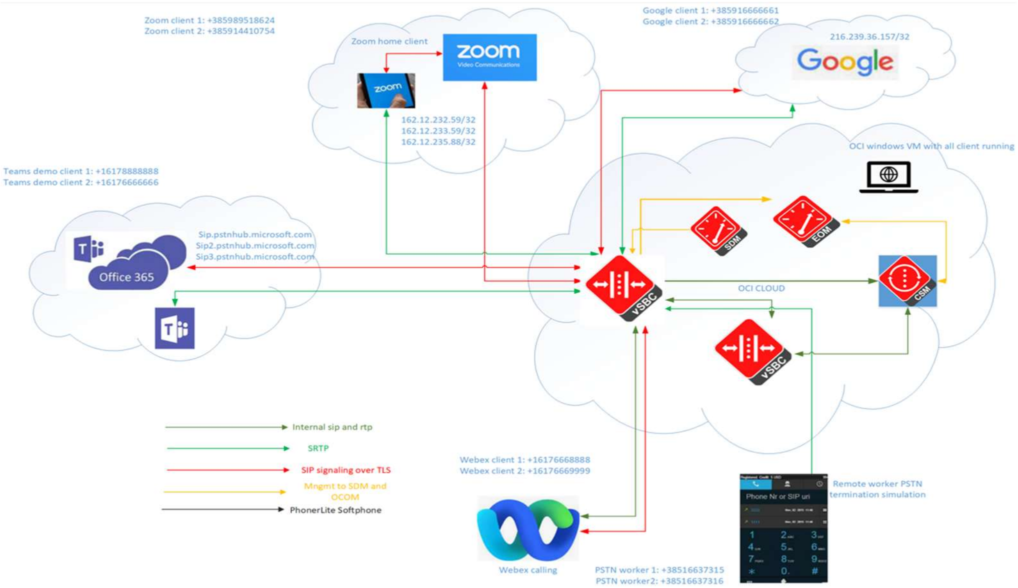
Software SBC - SCZ920.p3

Software OCOM – 5.2

Introduction

One of the main aspects with any UCaaS deployment is security as it comes mandatory for both SIP and RTP. Given the complexity this document will outline some of the best current practices starting to prepare SBC for UCaaS deployment, being however applicable, to any setup that involves TLS and SRTP

LAB UCaaS demo topology



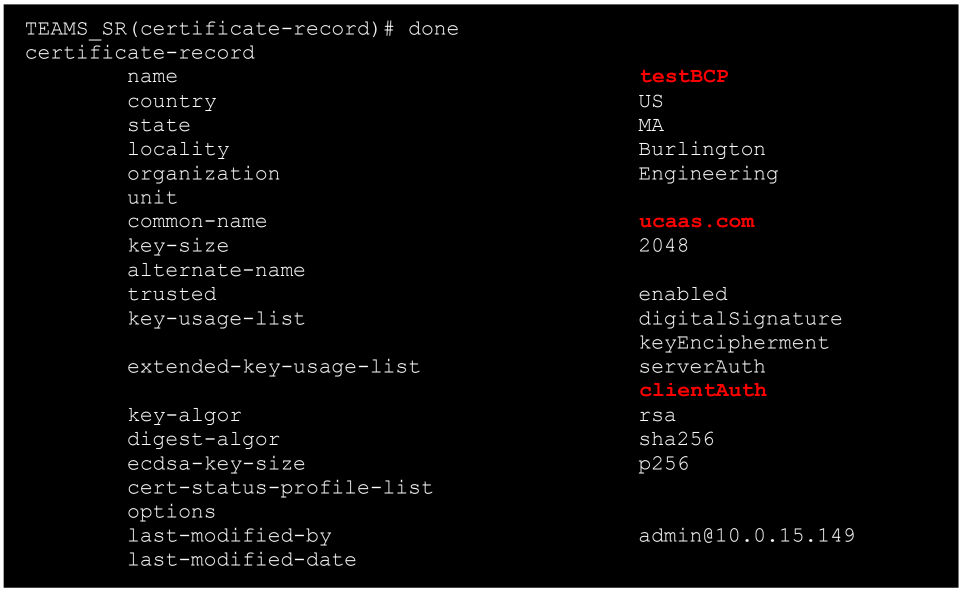
SBC security configuration objects

End entity certificate

Every UCaaS integration comes with mutual TLS as mandatory and preparation step one in SBC is to build its certificate-record end-entity certificate. In a nutshell this is certificate SBC is going to use to introduce itself during TLS handshake. With mutual TLS, SBC will present this certificate acting as server as server certificate or it's going to answer with this certificate acting as client upon server's certificate request, in mutual TLS server requires client side authentication too. At present there are two models end-entity certificate can be created/loaded to SBC

End entity certificate install from SBC generated CSR

Generating end-entity certificate starts with certificate-record creation in SBC's main security configuration branch. As highlighted below and bolded red one might see parameters that are mandatory – name, common-name(allocated SBC domain that will be protected) and optionally some extension flags(other parameters as equally important and are to be aligned between 2 ends terminating TLS). In this use case additional extension configured is client-auth as it comes mandatory with mutual TLS and CSR that will be created based on certificate record will carry out a request to support this extension. Remark here that CSR desired extension flags may be modified, removed or added by certificate signing authority. No matter correct CSR generation, signed certificate should be checked for all extension flags that are expected. That said it's obvious that wrongly signed, certificate may end up without client-auth flag that will prevent mutual TLS handshake to work. It is important in this process to be aligned with CA on what flags signed certificate should inherit from CSR.

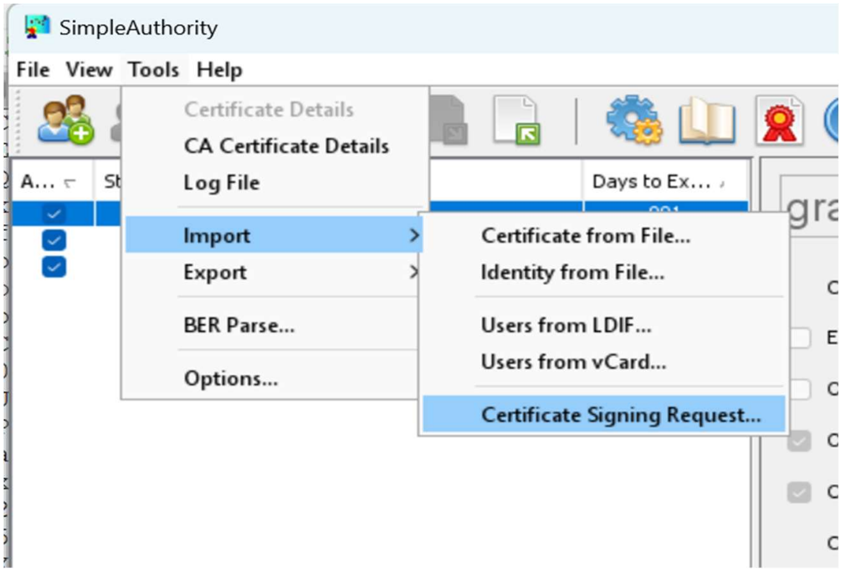


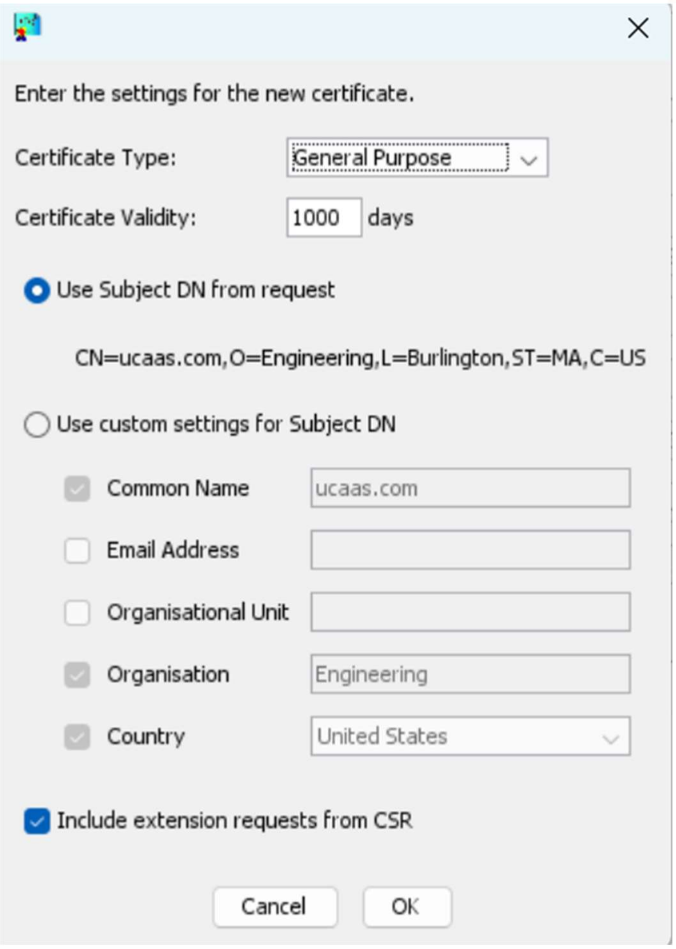
```
TEAMS_SR(certificate-record)# done
certificate-record
        name                                    testBCP
        country                                 US
        state                                   MA
        locality                                Burlington
        organization                            Engineering
        unit
        common-name                             ucaas.com
        key-size                                2048
        alternate-name
        trusted                                 enabled
        key-usage-list                          digitalSignature
                                                keyEncipherment
        extended-key-usage-list                 serverAuth
                                                clientAuth
        key-algor                               rsa
        digest-algor                            sha256
        ecdsa-key-size                          p256
        cert-status-profile-list
        options
        last-modified-by                        admin@10.0.15.149
        last-modified-date
```

This model of generating end-entity certificate starts with certificate-record object out of which one triggers CSR creation.

```
TEAMS_SR# generate-certificate-request testBCP
Generating Certificate Signing Request. This can take several minutes....

-----BEGIN CERTIFICATE REQUEST-----
MIIC2zCCAcMCAQAwWTELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAk1BMRMwEQYDVQQH
EwpCdXJsaW5ndG9uMRQwEgYDVQQKEwtFbmdpbmVlcmluZzESMBAGA1UEAxMJdWNh
YXMuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvAxT8EyHnC/J
IiA2Q3FuFM9MDdPa+7fnmLQq9r2nPTOBA3fOyV4fgdvGWZMJWB4F1OBrEC95pbLg
PL3KdXI1gTkoshyOBSbo31JWosvyABwgYXpopaZfBo0aGSvOO4ptgW+GW0V3XWge
oTZKNS9vFNGYU3ycYFpYGZIA3B52OXB+bb2l0hDvFccS7aqKo/kYas8JGSqtU88r
XZ/dFNMAew/bWYlxojbAJERBkBsb0AMDTIpeaT+yb6QZUYlc+BA4pjvcxKy2bXza
NrkSxbMM2Ekj6epuSPCjBwiJtwYU5Dio2VZlCbKqJ97QQOo0InxwBHO+GYolbqE2
iaO2EqesxQIDAQABoD0wOwYJKoZIhvcNAQkOMS4wLDALBgNVHQ8EBAMCBaAwHQYD
VR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMA0GCSqGSIb3DQEBCwUAA4IBAQCB
MAwQs1KUt8GvuanllWPFhNJAGOK9pNO85/zZzyM/Whd/fcCGjszPnnMghFmTMPTP
kHgCGfwunedQUj4hfBay7V+qtHkpRgYoAj9pVKnqZ9xQU0QtChiQM6p/nCTunTZ2
vCZxhTiU2gQW8VtRlRxZp/vqUTrPJS6NQMAy0eys69X+mq4KshimtjE181UONEDx
wcINPGjaTxW37CMSYz2+1vvpECN2Bbmub2a9BeWOTiGzNXwANNPPK8OPGQpGY3aT
ASLYRvRPa4PsxgS7xI5E5uuEELFW8r1qS/XYYfaz7VlBQjk29hneq5dAVdnDWQ10
zK51oukFQBCcC9Xqq5Xg
-----END CERTIFICATE REQUEST-----

WARNING: Configuration changed, run "save-config" command.
TEAMS_SR#
```

Next step is to supply certification authority with generated CSR to be signed and ported back to SBC. In this exercise I'll be using windows application "Simple authority" that acts as CA. Saving above output to a file I'm loading it to CA app for signature





Enter the settings for the new certificate.

Certificate Type: General Purpose

Certificate Validity: 1000 days

● Use Subject DN from request

CN=ucaas.com,O=Engineering,L=Burlington,ST=MA,C=US

○ Use custom settings for Subject DN

☑ Common Name: ucaas.com
☐ Email Address:
☐ Organisational Unit:
☑ Organisation: Engineering
☑ Country: United States

☑ Include extension requests from CSR

As you may note on the right hand side CA loads the CSR and presents data from it as we defined them in certificate-record configuration object. As to revert to previous discussion please note a thick on “Include extension requests from CSR”. This means that signing the certificate “client-auth” extension flag will remain as specified in CSR

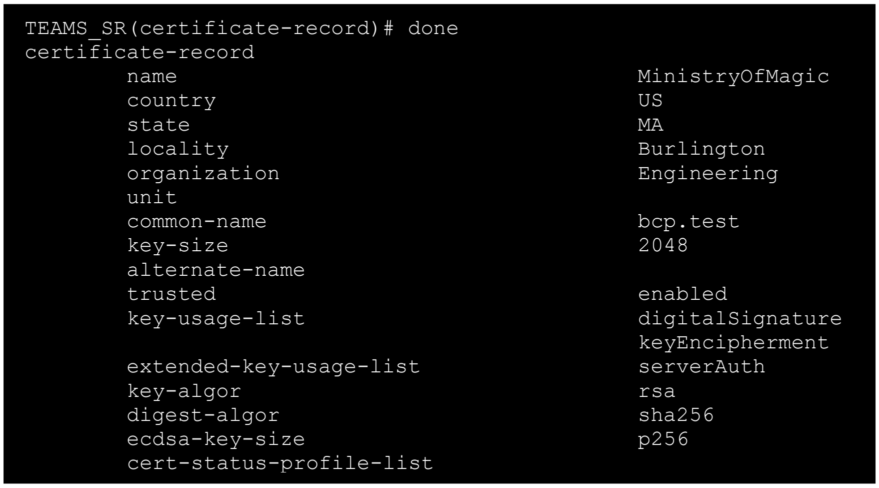
Hitting “ok” certificate is signed and content in pem format ready to be pasted back to SBC:

```
TEAMS_SR# import-certificate try-all testBCP
IMPORTANT:
        Terminate the certificate with ";" to exit.......
-----BEGIN CERTIFICATE-----
MIIDszCCApugAwIBAgIGAY2xNfQcMA0GCSqGSIb3DQEBCwUAMGExCzAJBgNVBAYT
AkhSMRQwEgYDVQQKDAtNYXRlaiBNYXJpYzEgMB4GA1UECwwXQ2VydGlmaWNhdGlv
biBBdXRob3JpdHkxGjAYBgNVBAMMEU1pbmlzdHJ5IG9mIE1hZ2ljMB4XDTI0MDIx
NjA5MTgzOVoXDTI2MTExMjA5MTgzOVowWTELMAkGA1UEBhMCVVMxCzAJBgNVBAgT
Ak1BMRMwEQYDVQQHEwpCdXJsaW5ndG9uMRQwEgYDVQQKEwtFbmdpbmVlcmluZzES
MBAGA1UEAxMJdWNhYXMuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEAvAxT8EyHnC/JIiA2Q3FuFM9MDdPa+7fnmLQq9r2nPTOBA3fOyV4fgdvGWZMJ
WB4F1OBrEC95pbLgPL3KdXI1gTkoshyOBSbo31JWosvyABwgYXpopaZfBo0aGSvO
O4ptgW+GW0V3XWgeoTZKNS9vFNGYU3ycYFpYGZIA3B52OXB+bb2l0hDvFccS7aqK
o/kYas8JGSqtU88rXZ/dFNMAew/bWYlxojbAJERBkBsb0AMDTIpeaT+yb6QZUYlc
+BA4pjvcxKy2bXzaNrkSxbMM2Ekj6epuSPCjBwiJtwYU5Dio2VZlCbKqJ97QQOo0
InxwBHO+GYolbqE2iaO2EqesxQIDAQABo3kwdzAfBgNVHSMEGDAWgBTomkZwyHuA
cxUwIRcA6EUWH4GrKTAJBgNVHRMEAjAAMAsGA1UdDwQEAwIFoDAdBgNVHQ4EFgQU
wfsMIJDqwhxgjripa5jxJsY7OdowHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUF
BwMCMA0GCSqGSIb3DQEBCwUAA4IBAQAiqQ44CXkLaRwVEmjfdPvOySip0e+XNxk
VBQJVKyi2SfCLsXpK0lF8VD2KuR2ue90/GBYBOGyVD0a6xCpIl434uVGxsUG2ubO
UtDtRW0IPnzYTpI/eeEoqVVTr9RqH98aPYvLWf6kym3N3ejS3fF+Lu/M77UlXiSg
EXdnGkUoADrD6tYi0FsE6rLBgWyr2pPORr+H30UHNjr45y1R6CXo0p8OSTYz6TR2
lCm5gnxcyDHNryaD3ZdtI/7CV1Xiq4IToVmwrTDJpxgfSoIQ1nLHfgAgjHlp+wHx
QVr5dl2QPqaMuC0TinKb5WkM5i9fuNFwUj0GV2fuPIBk3wY/F6Oc
-----END CERTIFICATE-----
```

Only precise verification of what has been ported back after save&activate we get executing “show security certificate-record detail/brief”

```
certificate-record: testBCP
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1708075119644 (0x18db135f41c)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer:
            C=HR
            O=Matej Maric
            OU=Certification Authority
            CN=Ministry of Magic
        Validity
            Not Before: Feb 16 09:18:39 2024 GMT
            Not After : Nov 12 09:18:39 2026 GMT
        Subject:
            C=US
            ST=MA
            L=Burlington
            O=Engineering
            CN=ucaas.com
        X509v3 extensions:
            X509v3 Authority Key Identifier:

keyid:E8:9A:46:70:C8:7B:80:73:15:30:21:17:00:E8:45:16:1F:81:AB:2
9

            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature, Key Encipherment
            X509v3 Subject Key Identifier:

C1:FB:0C:20:90:EA:C2:1C:60:8E:B8:A9:6B:98:F1:26:C6:3B:39:DA
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client
Authentication
```

Exchanging certificates in TLS handshake one must provide a full signing chain and not only signed certificate. For this reason we need to load in SBC also public certificate of authority that signed our CSR. For this purpose we will create another certificate-record as outlined below. Very important remark here is that there is a big difference between certificate record created to build end-entity certificate and the one we built below to load CA public certificate. First one was associated with unique private key given the CSR creation and only signed cert matching the private key is suitable to be loaded back. Latter one below is not associated with any private key and SBC will load there any CA public certificate overwriting default SBC certificate-record content. CA public(root and intermediates) certificates are public and can be easily fetched from internet.

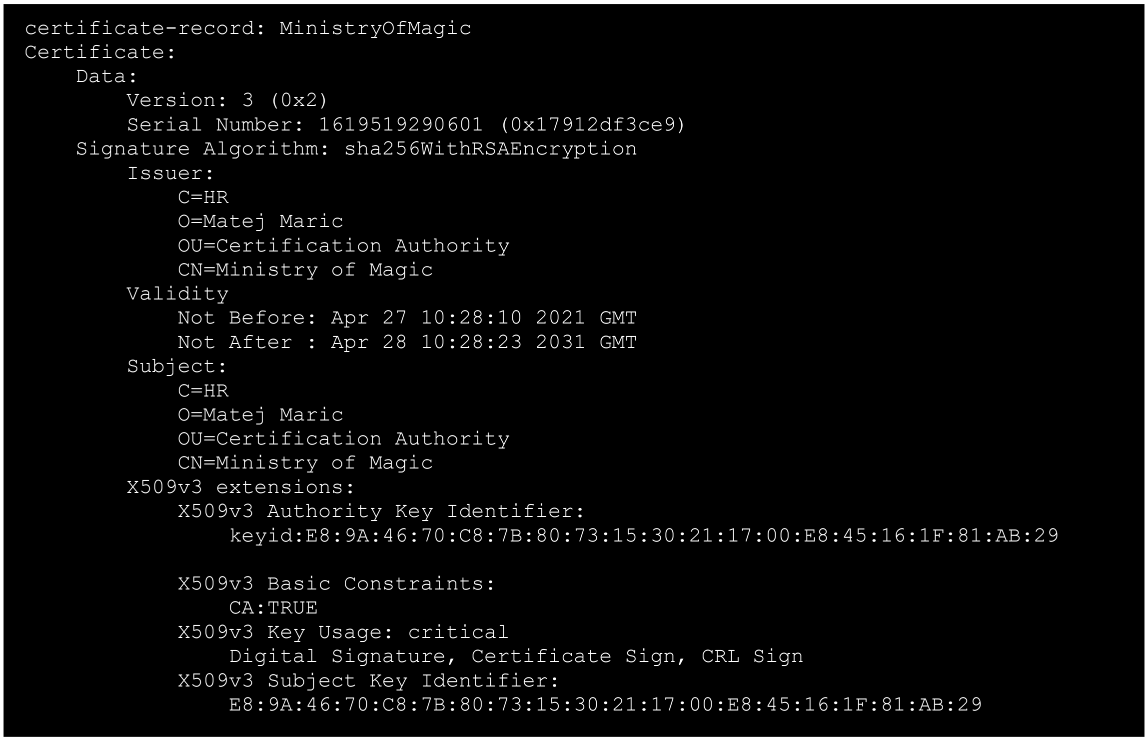


```
TEAMS_SR(certificate-record)# done
certificate-record
        name                                    MinistryOfMagic
        country                                 US
        state                                   MA
        locality                                Burlington
        organization                            Engineering
        unit
        common-name                             bcp.test
        key-size                                2048
        alternate-name
        trusted                                 enabled
        key-usage-list                          digitalSignature
                                                keyEncipherment
        extended-key-usage-list                 serverAuth
        key-algor                               rsa
        digest-algor                            sha256
        ecdsa-key-size                          p256
        cert-status-profile-list
```

As said content above is irrelevant loading the CA public certs and I will just load my CA Root certificate over this certificate-record. To emphasize that this step must be repeated in case there are intermediate certificates in CA signing chain.

```
TEAMS_SR# import-certificate try-all MinistryOfMagic
IMPORTANT:
        Please enter the certificate in the PEM format.
        Terminate the certificate with ";" to exit.......
-----BEGIN CERTIFICATE-----
MIIDojCCAoqgAwIBAgIGAXkS3zzpMA0GCSqGSIb3DQEBCwUAMGExCzAJBgNVBAYT
AkhSMRQwEgYDVQQKDAtNYXRlaiBNYXJpYzEgMB4GA1UECwwXQ2VydGlmaWNhdGlv
biBBdXRob3JpdHkxGjAYBgNVBAMMEU1pbmlzdHJ5IG9mIE1hZ2ljMB4XDTIxMDQy
NzEwMjgxMFoXDTMxMDQyODEwMjgyM1owYTELMAkGA1UEBhMCSFIxFDASBgNVBAoM
C01hdGVqIE1hcmljMSAwHgYDVQQLDBdDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTEa
MBgGA1UEAwwRTWluaXN0cnkgb2YgTWFnaWMwggEiMA0GCSqGSIb3DQEBAQUAA4IB
DwAwggEKAoIBAQDJSsPHH3PjBBJWt/fz+6WWZrGmJ7W4WyjujxD85yD/FJDatZ2v
Tbdk+sOop8sbcZt3bNNulNUfL861S3yMjkTnC5lpStVjs1W9yNJSkgRv7pEZR5i6
5BaEJg48J8puBwB5qY1JhZZjruGkhTo7RiYGxjv40jp8tfFaVPt7c7t6YOmaP+34
zGrGzGvWEH4WTDGY8EbUTWnZbg2YUUAVsniUDPn9ohyqm/YoW+JZBQ2a9JyJA8uu
weijgD7lZnewxlqzGYs018zqbcs//VClxbHaDiiStUCjwGtsGiUDdCk8OI7v3yJC
N+8lYgifFOWy4oACGOMUfNzQKaYEzDxecOn/AgMBAAGjYDBeMB8GA1UdIwQYMBaA
FOiaRnDIe4BzFTAhFwDoRRYfgaspMAwGA1UdEwQFMAMBAf8wDgYDVR0PAQH/BAQD
AgGGMB0GA1UdDgQWBBTomkZwyHuAcxUwIRcA6EUWH4GrKTANBgkqhkiG9w0BAQsF
AAOCAQEAffURW2IxwwssBtmkjItDFEytAwPpyez2a+g8e10i6Huzu/i/Kbj3YnZJ
lBDH5mCYwaqs9L+WpRswFSCVMm4hFaB5L4UOR3omznLJXgP+TvqzqU8o0H8XVirB
BmyUQ4QWsfrzsmIQAXPuyVfsuhdpNPc3ojOLhluyOOZse0y1vWpoaVmKLpkRS8+b
ewx2gOwKMpqxD6iF+Q9cXSbtLVIL2h6fjvaWBeQMVxK88cQO7zaBiKNXRIPLJ/+U
VIAkqagYKp6/R3Uh+1G2Mz1mhyZMZN/+qxOAl2D5lGYLLq+nVBbtpgSQBo6mZEfA
5Mk75zfGhEnDtAQ9sYmcNtkSKA5cqA==
-----END CERTIFICATE-----
Certificate imported successfully....
WARNING: Configuration changed, run "save-config" command.
```

Proper verification kicks in again with “show security certificate-record detail/brief”. It will expose details of newly loaded CA public certificate. Please note below that content of this record has nothing to do anymore with default configuration we have put in public CA certificate-record object.



```
certificate-record: MinistryOfMagic
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1619519290601 (0x17912df3ce9)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer:
            C=HR
            O=Matej Maric
            OU=Certification Authority
            CN=Ministry of Magic
        Validity
            Not Before: Apr 27 10:28:10 2021 GMT
            Not After : Apr 28 10:28:23 2031 GMT
        Subject:
            C=HR
            O=Matej Maric
            OU=Certification Authority
            CN=Ministry of Magic
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:E8:9A:46:70:C8:7B:80:73:15:30:21:17:00:E8:45:16:1F:81:AB:29

            X509v3 Basic Constraints:
                CA:TRUE
            X509v3 Key Usage: critical
                Digital Signature, Certificate Sign, CRL Sign
            X509v3 Subject Key Identifier:
                E8:9A:46:70:C8:7B:80:73:15:30:21:17:00:E8:45:16:1F:81:AB:29
```

End entity certificate install from PKCS12 bundle

PKCS12 is a bundle that consists of private key and signed certificate material. In other word it’s elsewhere generated end-entity certificate that SBC supports. In this approach there is no need for CSR generation in SBC as SBC is going to load signed certificate and associate with the private key being part of the same bundle. Also there is no need to create certificate-record object as it’s going to be automatically created by SBC upon loading the p12 file.

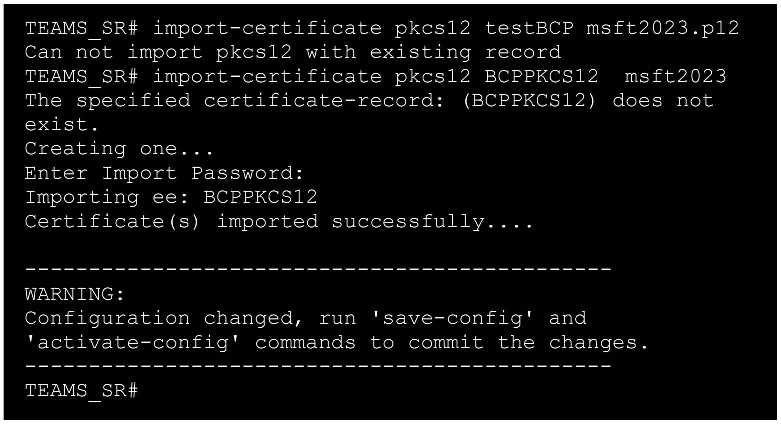
In practice this means that our customers may be supplied by their security team with a file typically carrying .pfx or p12 extension. Such file needs to be put in /opt folder before attempted to be loaded to SBC. Only issue detected in field trying to upload pkcs12 form is in the way bundle was created and if

SBC prompts an error trying to load such a bundle it could be pkcs12 has to be re-created(openssl) as outlined below

openssl pkcs12 -in <filename>.pfx -nocerts -out key.pem (extracts private key from bundle)
openssl pkcs12 -in <filename>.pfx -clcerts -nokeys -out cert.pem (extracts signed cert from bundle)

With these two outputs we will re-create pkcs12 bundle using PBE-SHA1-3DES as it is only one SBC today supports.
openssl pkcs12 -keypbe PBE-SHA1-3DES -certpbe PBE-SHA1-3DES -export -out msft2023.p12 -inkey key.pem -in cert.pem

At this point msft2023.p12 should be properly formatted and ready to be ported into SBC.



```
TEAMS_SR# import-certificate pkcs12 testBCP msft2023.p12
Can not import pkcs12 with existing record
TEAMS_SR# import-certificate pkcs12 BCPPKCS12  msft2023
The specified certificate-record: (BCPPKCS12) does not
exist.
Creating one...
Enter Import Password:
Importing ee: BCPPKCS12
Certificate(s) imported successfully....

----------------------------------------------
WARNING:
Configuration changed, run 'save-config' and
'activate-config' commands to commit the changes.
----------------------------------------------
TEAMS_SR#
```

Please note that certificate initially created with CSR in SBC may be also exported from SBC in pkcs12 form and loaded to multiple SBCs. This however depends on exact customer deployment and may have security implication given it is same private key re-used in multiple devices. This will not be discussed deeper as part of this BCP.

TLS profile

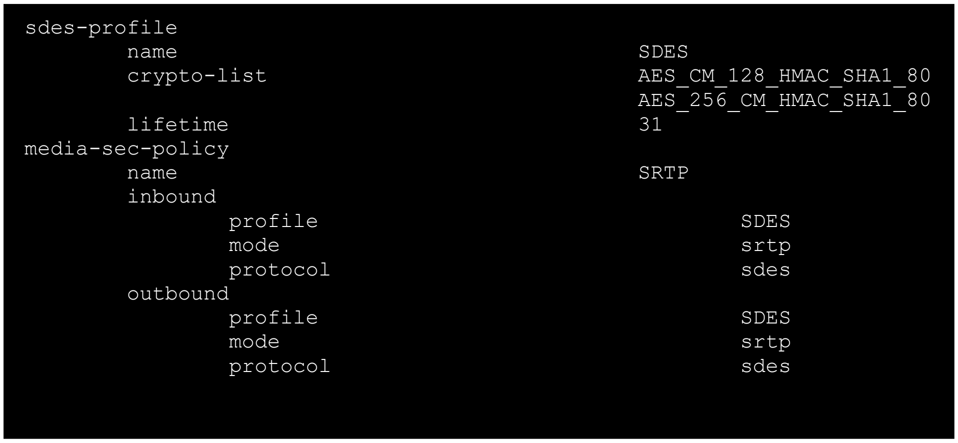
Getting done with certificate creation with need to assign certificate-records properly in tls-profile configuration element. Such tls-profile is later assigned to sip-interface configuration object

```
TEAMS_SR# sho configuration tls-profile
tls-profile
        name                                    TEAMS
        end-entity-certificate                  teams2023
        trusted-ca-certificates                 GoDaddyCA
                                                Baltimore
                                                DigiCertRootG2
        cipher-list                             ALL
        verify-depth                            10
        mutual-authenticate                     enabled
        tls-version                             tlsv12
        options
        cert-status-check                       disabled
        cert-status-profile-list
        ignore-dead-responder                   disabled
        allow-self-signed-cert                  disabled
```

Highlighting here that mutual-authenticate parameter, in UCaaS use case, must be enabled otherwise SBC as a server will not request client certificate in TLS handshake and acting as client it will fail to send its certificate upon server’s request. Whilst end-entity parameter looks clear on how to be configured trusted-ca-certificates must consist of the local CA chain. In other words there we must specify all intermediates and root CA of local certificate as the whole chain must be presented in TLS handshake. Also, there we must specify roof top Root CA of remote party, certificate record of remote party CA shall be created in same fashion as for local CA, please refer to chapter “End entity certificate” public CA certs loading paragraphs.

SDES profile and media-security policy

Configuring two additional configuration objects in SBC we should cover SRTP negotiation and termination on legs where SRTP is mandatory, typically only legs towards UCaaS vendor. Below the sample configuration whereas media-sec-profile gets attached to UCaaS realms.



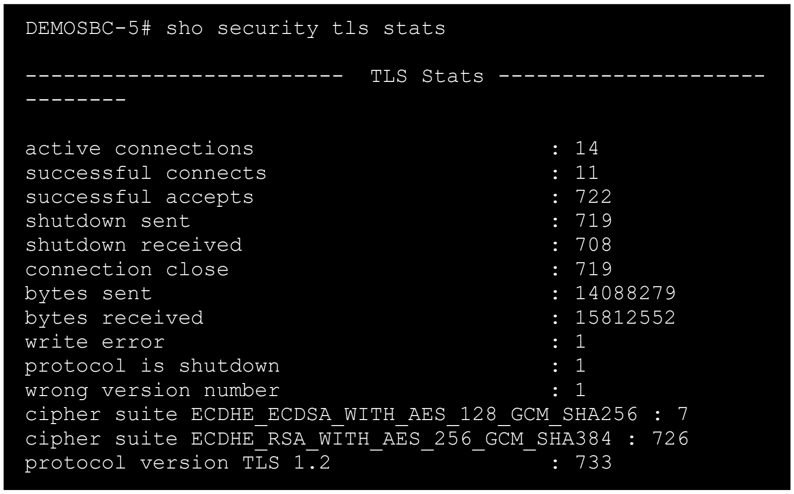
```
sdes-profile
        name                                    SDES
        crypto-list                             AES_CM_128_HMAC_SHA1_80
                                                AES_256_CM_HMAC_SHA1_80
        lifetime                                31
media-sec-policy
        name                                    SRTP
        inbound
                profile                                 SDES
                mode                                    srtp
                protocol                                sdes
        outbound
                profile                                 SDES
                mode                                    srtp
                protocol                                sdes
```

We should be now all set to test our application.

TLS and SRTP troubleshoot

In order to verify TLS connection against remote agents works fine same steps might be followed as for any other TCP/UDP agents. SA configured to be SIP OPTIONS pinged will appear in status active in case OPTIONS are successfully replied and if SIP OPTIONS are successfully replied it means underlying transport TLS handshake went well too. Sipmsg.log may give insights in sip message details as well as calls might be checked in OCOM supplied with data from embedded probe(SBC acts as a probe). SBC sends decrypted data to OCOM mediation engine. Calls are then easily readable in form of ladder diagrams.

Successful TLS and SRTP verification

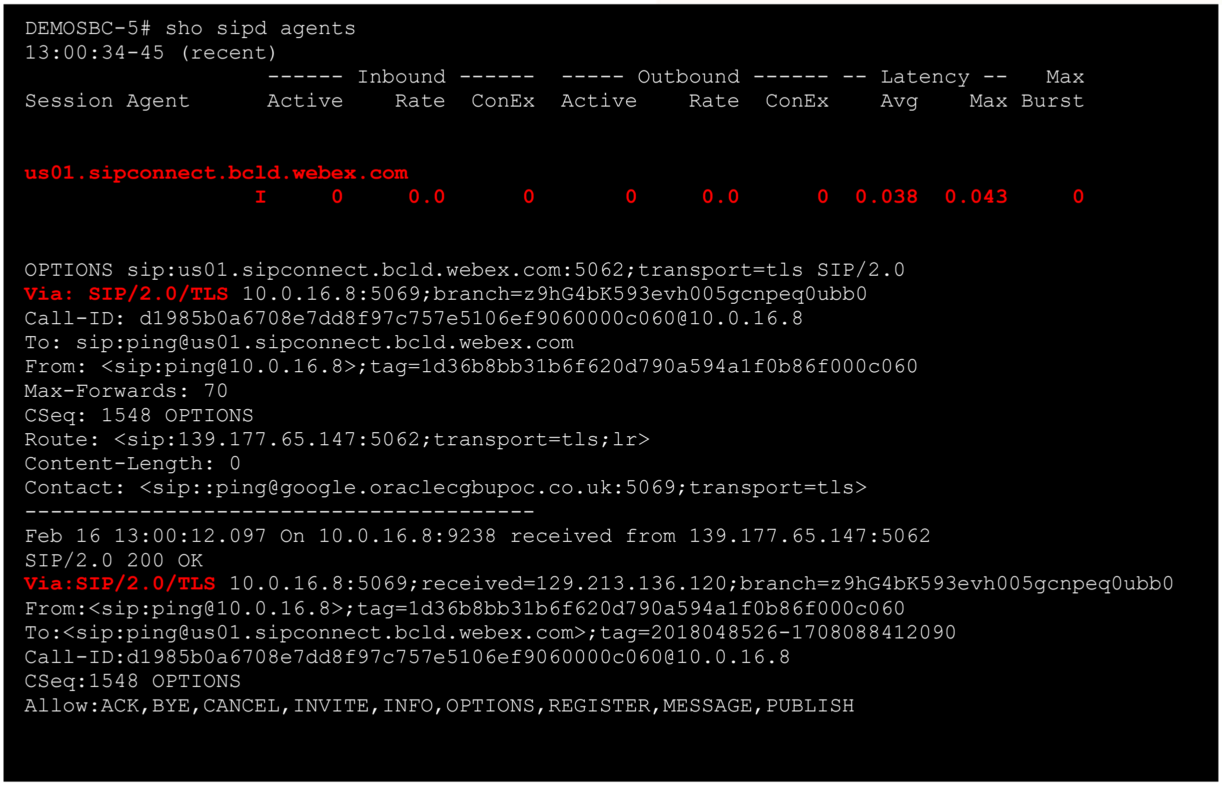


```
DEMOSBC-5# sho security tls stats

-------------------------  TLS Stats ---------------------
--------

active connections                        : 14
successful connects                       : 11
successful accepts                        : 722
shutdown sent                             : 719
shutdown received                         : 708
connection close                          : 719
bytes sent                                : 14088279
bytes received                            : 15812552
write error                               : 1
protocol is shutdown                      : 1
wrong version number                      : 1
cipher suite ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 : 7
cipher suite ECDHE_RSA_WITH_AES_256_GCM_SHA384 : 726
protocol version TLS 1.2                  : 733
```

TLS security stats will expose overall number of active and closed TLS connections along with per chipper and tls version stats.

Successful TLS session-agent verification from CLI:

Best current practice

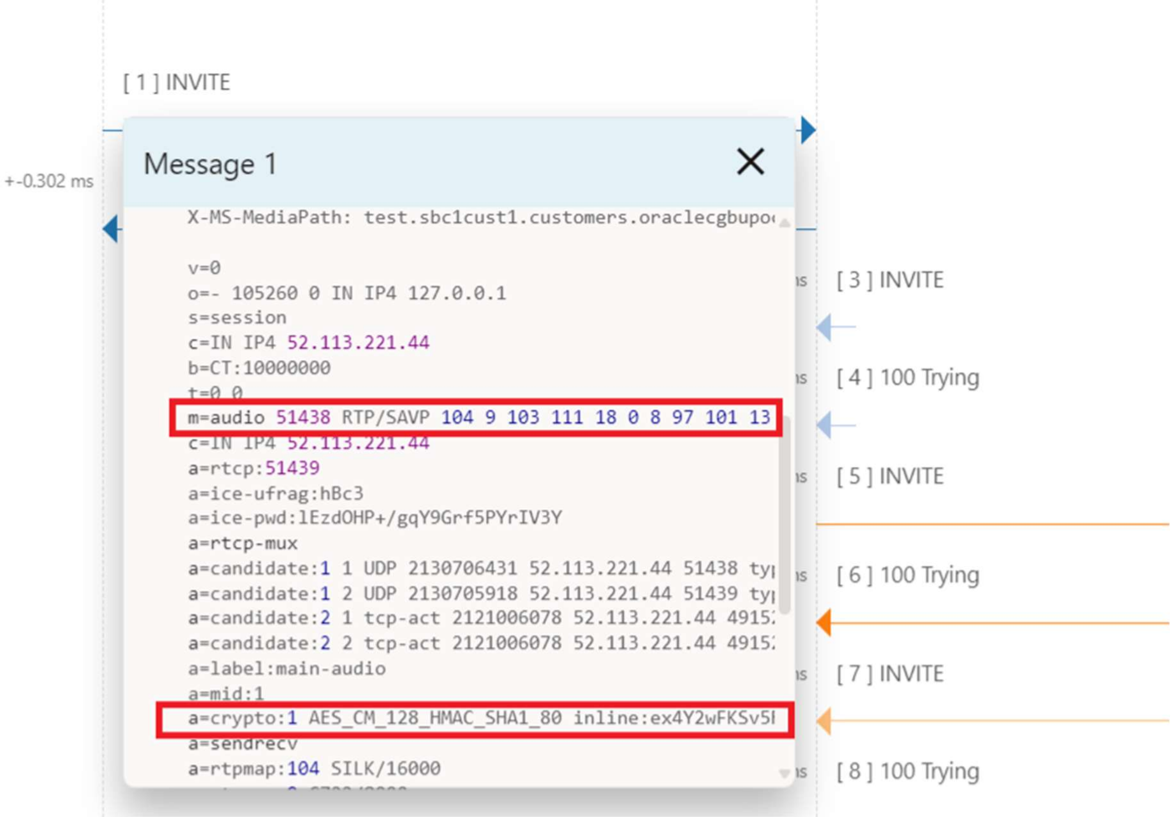


```
DEMOSBC-5# sho sipd agents
13:00:34-45 (recent)
                   ------ Inbound ------  ----- Outbound ------ -- Latency --   Max
Session Agent      Active    Rate  ConEx  Active    Rate  ConEx    Avg     Max Burst


us01.sipconnect.bcld.webex.com
                  I     0     0.0      0       0     0.0      0  0.038  0.043      0


OPTIONS sip:us01.sipconnect.bcld.webex.com:5062;transport=tls SIP/2.0
Via: SIP/2.0/TLS 10.0.16.8:5069;branch=z9hG4bK593evh005gcnpeq0ubb0
Call-ID: d1985b0a6708e7dd8f97c757e5106ef9060000c060@10.0.16.8
To: sip:ping@us01.sipconnect.bcld.webex.com
From: <sip:ping@10.0.16.8>;tag=1d36b8bb31b6f620d790a594a1f0b86f000c060
Max-Forwards: 70
CSeq: 1548 OPTIONS
Route: <sip:139.177.65.147:5062;transport=tls;lr>
Content-Length: 0
Contact: <sip::ping@google.oraclecgbupoc.co.uk:5069;transport=tls>
----------------------------------------
Feb 16 13:00:12.097 On 10.0.16.8:9238 received from 139.177.65.147:5062
SIP/2.0 200 OK
Via:SIP/2.0/TLS 10.0.16.8:5069;received=129.213.136.120;branch=z9hG4bK593evh005gcnpeq0ubb0
From:<sip:ping@10.0.16.8>;tag=1d36b8bb31b6f620d790a594a1f0b86f000c060
To:<sip:ping@us01.sipconnect.bcld.webex.com>;tag=2018048526-1708088412090
Call-ID:d1985b0a6708e7dd8f97c757e5106ef9060000c060@10.0.16.8
CSeq:1548 OPTIONS
Allow:ACK,BYE,CANCEL,INVITE,INFO,OPTIONS,REGISTER,MESSAGE,PUBLISH
```

Successful TLS call verification in OCOM:

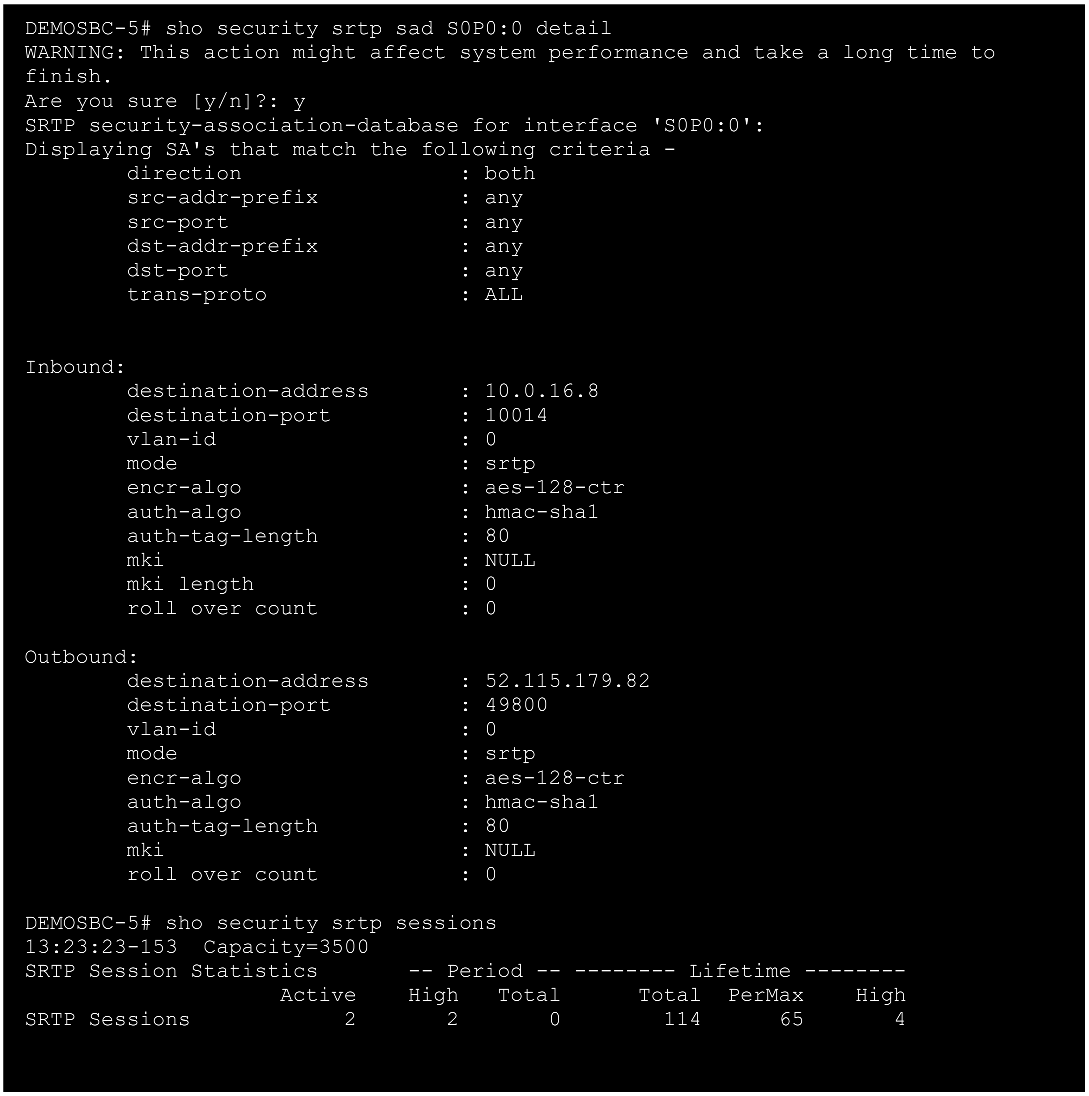
```
INVITE sip:+38516637345@10.0.16.8:5083;user=phone SIP/2.0
FROM: "econnect"<sip:+17813131033@sip.pstnhub.microsoft.com:5061;
TO: <sip:+38516637345@sbc1cust1.customers.oraclecgbupoc.co.uk:506
CSEQ: 1 INVITE
CALL-ID: f72c7fa3f62e5b07907dff753fbe9772
MAX-FORWARDS: 69
Via: SIP/2.0/TCP 10.0.17.35:5080;branch=z9hG4bK[illegible]5s9pug
VIA: SIP/2.0/TLS 52.114.132.46:5061;branch=z9hG4bK514c95ce
Record-Route: <sip:[illegible]f9p6fv
Record-Route: <sip:SD3qcpb+jiq16ba9kico9d8rrca1c36ru-5b9peffjfuvp
RECORD-ROUTE: <sip:sip-du-a-us.pstnhub.microsoft.com:5061;transpo
CONTACT: <sip:api-du-b-euno.pstnhub.microsoft.com:443;x-i=04f3622
```

SRTP is negotiated within SIP TLS encrypted connection with SDP exchange. SRTP session is considered successful if there is a match in crypto list algorithms between 2 parties.



```
X-MS-MediaPath: test.sbc1cust1.customers.oraclecgbupo

v=0
o=- 105260 0 IN IP4 127.0.0.1
s=session
c=IN IP4 52.113.221.44
b=CT:10000000
t=0 0
m=audio 51438 RTP/SAVP 104 9 103 111 18 0 8 97 101 13
c=IN IP4 52.113.221.44
a=rtcp:51439
a=ice-ufrag:hBc3
a=ice-pwd:1EzdOHP+/gqY9Grf5PYrIV3Y
a=rtcp-mux
a=candidate:1 1 UDP 2130706431 52.113.221.44 51438 ty
a=candidate:1 2 UDP 2130705918 52.113.221.44 51439 ty
a=candidate:2 1 tcp-act 2121006078 52.113.221.44 4915
a=candidate:2 2 tcp-act 2121006078 52.113.221.44 4915
a=label:main-audio
a=mid:1
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:ex4Y2wFKSv5
a=sendrecv
a=rtpmap:104 SILK/16000
```

Upon a successful SRTP call established SRTP security associations may be displayed in CLI:



```
DEMOSBC-5# sho security srtp sad S0P0:0 detail
WARNING: This action might affect system performance and take a long time to
finish.
Are you sure [y/n]?: y
SRTP security-association-database for interface 'S0P0:0':
Displaying SA's that match the following criteria -
        direction                  : both
        src-addr-prefix            : any
        src-port                   : any
        dst-addr-prefix            : any
        dst-port                   : any
        trans-proto                : ALL


Inbound:
        destination-address        : 10.0.16.8
        destination-port           : 10014
        vlan-id                    : 0
        mode                       : srtp
        encr-algo                  : aes-128-ctr
        auth-algo                  : hmac-sha1
        auth-tag-length            : 80
        mki                        : NULL
        mki length                 : 0
        roll over count            : 0

Outbound:
        destination-address        : 52.115.179.82
        destination-port           : 49800
        vlan-id                    : 0
        mode                       : srtp
        encr-algo                  : aes-128-ctr
        auth-algo                  : hmac-sha1
        auth-tag-length            : 80
        mki                        : NULL
        roll over count            : 0

DEMOSBC-5# sho security srtp sessions
13:23:23-153  Capacity=3500
SRTP Session Statistics       -- Period -- -------- Lifetime --------
                    Active    High    Total      Total  PerMax    High
SRTP Sessions            2       2        0        114      65       4
```

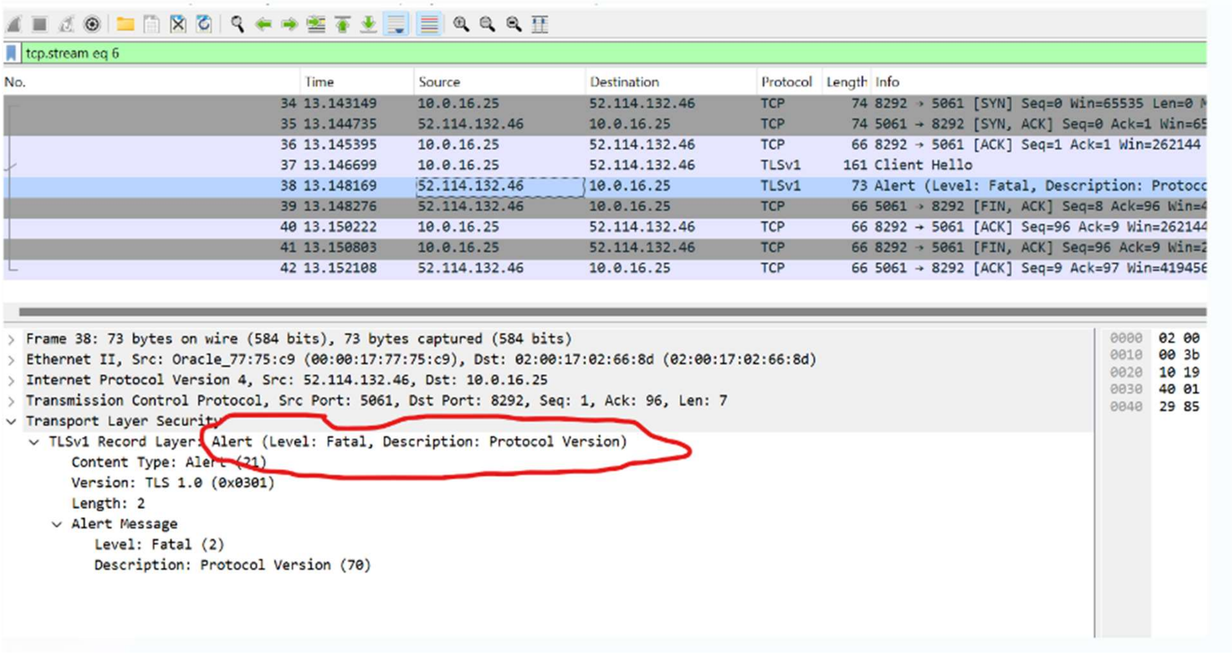
Failing TLS and SRTP cases

Failures in TLS handshake may be observed in log.atcpd on debug log level but it’s probably the easiest to troubleshoot setting up packet-trace local in SBC and viewing captured data in wireshark as most of failures pop up in a stage we still may see traffic in clear. TLS handshake may fail for couple of reasons, highlighting common ones in UCaaS environment:

- TLS version mismatch
- TLS cipher suite mismatch
- Certificate record issues

TLS 1.2 and 1.3 are commonly used today and 1.0 and 1.1 became deprecated. It's client that starts TLS handshake with "Client Hello" and indicates its TLS version and cipher suite support. If connection terminates without "Server Hello" then either TLS version or cipher suit does not match server side.

Screenshot below is packet trace local presentation where TLS was attempted from SBC simulating only TLS1.0 version support, remote are MSFT TLS session agents:



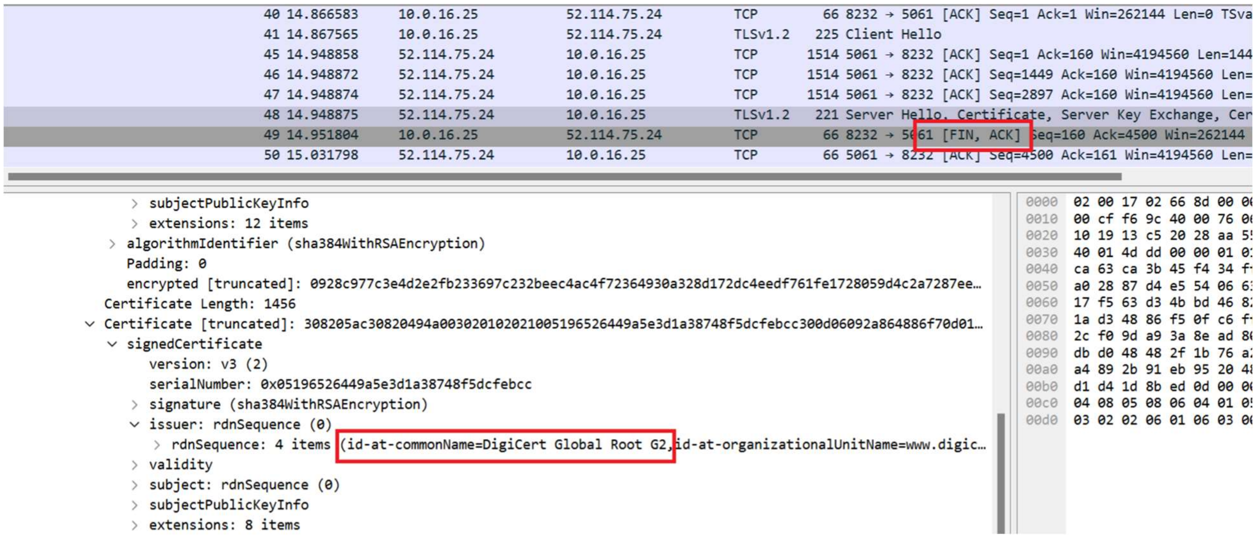
```
tcp.stream eq 6
No.    Time       Source          Destination     Protocol Length Info
    34 13.143149  10.0.16.25      52.114.132.46   TCP        74 8292 → 5061 [SYN] Seq=0 Win=65535 Len=0 M
    35 13.144735  52.114.132.46   10.0.16.25      TCP        74 5061 → 8292 [SYN, ACK] Seq=0 Ack=1 Win=65
    36 13.145395  10.0.16.25      52.114.132.46   TCP        66 8292 → 5061 [ACK] Seq=1 Ack=1 Win=262144
    37 13.146699  10.0.16.25      52.114.132.46   TLSv1     161 Client Hello
    38 13.148169  52.114.132.46   10.0.16.25      TLSv1      73 Alert (Level: Fatal, Description: Protocc
    39 13.148276  52.114.132.46   10.0.16.25      TCP        66 5061 → 8292 [FIN, ACK] Seq=8 Ack=96 Win=4
    40 13.150222  10.0.16.25      52.114.132.46   TCP        66 8292 → 5061 [ACK] Seq=96 Ack=9 Win=262144
    41 13.150803  10.0.16.25      52.114.132.46   TCP        66 8292 → 5061 [FIN, ACK] Seq=96 Ack=9 Win=2
    42 13.152108  52.114.132.46   10.0.16.25      TCP        66 5061 → 8292 [ACK] Seq=9 Ack=97 Win=419456

> Frame 38: 73 bytes on wire (584 bits), 73 bytes captured (584 bits)
> Ethernet II, Src: Oracle_77:75:c9 (00:00:17:77:75:c9), Dst: 02:00:17:02:66:8d (02:00:17:02:66:8d)
> Internet Protocol Version 4, Src: 52.114.132.46, Dst: 10.0.16.25
> Transmission Control Protocol, Src Port: 5061, Dst Port: 8292, Seq: 1, Ack: 96, Len: 7
v Transport Layer Security
  v TLSv1 Record Layer: Alert (Level: Fatal, Description: Protocol Version)
      Content Type: Alert (21)
      Version: TLS 1.0 (0x0301)
      Length: 2
    v Alert Message
        Level: Fatal (2)
        Description: Protocol Version (70)

0000  02 00
0010  00 3b
0020  10 19
0030  40 01
0040  29 85
```

In such a case, as mentioned earlier server side will answer "client hello" message straight with error before issuing "server hello" message. This guides us to check and correct tls-profile with proper TLS version and cipher suite supported by both parties. Log.atcpd the will reflect this failure as printed below:

```
Jul  9 14:15:27.930 [SERVICE] (0) TLS Handshake: client <<< TLS 1.0 Alert[length 0002], fatal protocol_version

Jul  9 14:15:27.930 [SERVICE] (0) <tlsengine.cpp:1549> SSL3 alert read:fatal:protocol version

Jul  9 14:15:27.930 [SERVICE] (0) <tlsengine.cpp:1567> SSL_connect:error in error
```

```
Jul  9 14:15:27.930 [SERVICE] (0) <tlsengine.cpp:4237> TLSEngine::TLSMachineDOControl, appData_m = 0, n = -1

Jul  9 14:15:27.930 [MINOR] (0) SSL_accept failed, fatal alert sent

Jul  9 14:15:27.930 [MINOR] (0) OpenSSL Error:error:1409442E:SSL routines:ssl3_read_bytes:tlsv1 alert protocol version:ssl/record/rec_layer_s3.c:1551:SSL alert number 70

Jul  9 14:15:27.930 [SERVICE] (0) <tlsengine.cpp:4357> TLSEngine::TLSMachineDOControl, appData_m = 0, retCode=32

Jul  9 14:15:27.930 [MINOR] (0) ServiceSocketProxyAdapter TCP:10.0.16.25:34348->52.114.75.24:5061 CheckAndRecvTLS, TLS Recv failed retCode: 32:TLS engine accept/connect failed on fd -1

Jul  9 14:15:27.930 [SERVICE] (0) <ServiceSocketProxyAdapter.cpp:1894> ServiceSocketProxyAdapter::Disconnect(void) (0x81bc3400)
```

Sorting out version and cipher suite there are certificates to be exchanged. Post “server hello” it is server presenting its certificate chain. Below failure simulation occurs when I remove public CA of remote party from tls-profile trusted-ca-certificate:



```
40 14.866583   10.0.16.25     52.114.75.24   TCP       66 8232 → 5061 [ACK] Seq=1 Ack=1 Win=262144 Len=0 TSva
41 14.867565   10.0.16.25     52.114.75.24   TLSv1.2  225 Client Hello
45 14.948858   52.114.75.24   10.0.16.25     TCP     1514 5061 → 8232 [ACK] Seq=1 Ack=160 Win=4194560 Len=144
46 14.948872   52.114.75.24   10.0.16.25     TCP     1514 5061 → 8232 [ACK] Seq=1449 Ack=160 Win=4194560 Len=
47 14.948874   52.114.75.24   10.0.16.25     TCP     1514 5061 → 8232 [ACK] Seq=2897 Ack=160 Win=4194560 Len=
48 14.948875   52.114.75.24   10.0.16.25     TLSv1.2  221 Server Hello, Certificate, Server Key Exchange, Cer
49 14.951804   10.0.16.25     52.114.75.24   TCP       66 8232 → 5061 [FIN, ACK] Seq=160 Ack=4500 Win=262144
50 15.031798   52.114.75.24   10.0.16.25     TCP       66 5061 → 8232 [ACK] Seq=4500 Ack=161 Win=4194560 Len=
```

```
      > subjectPublicKeyInfo
      > extensions: 12 items
    > algorithmIdentifier (sha384WithRSAEncryption)
      Padding: 0
      encrypted [truncated]: 0928c977c3e4d2e2fb233697c232beec4ac4f72364930a328d172dc4eedf761fe1728059d4c2a7287ee…
    Certificate Length: 1456
  v Certificate [truncated]: 308205ac30820494a003020102021005196526449a5e3d1a38748f5dcfebcc300d06092a864886f70d01…
    v signedCertificate
        version: v3 (2)
        serialNumber: 0x05196526449a5e3d1a38748f5dcfebcc
      > signature (sha384WithRSAEncryption)
      v issuer: rdnSequence (0)
        > rdnSequence: 4 items (id-at-commonName=DigiCert Global Root G2,id-at-organizationalUnitName=www.digic…
      > validity
      > subject: rdnSequence (0)
      > subjectPublicKeyInfo
      > extensions: 8 items
```

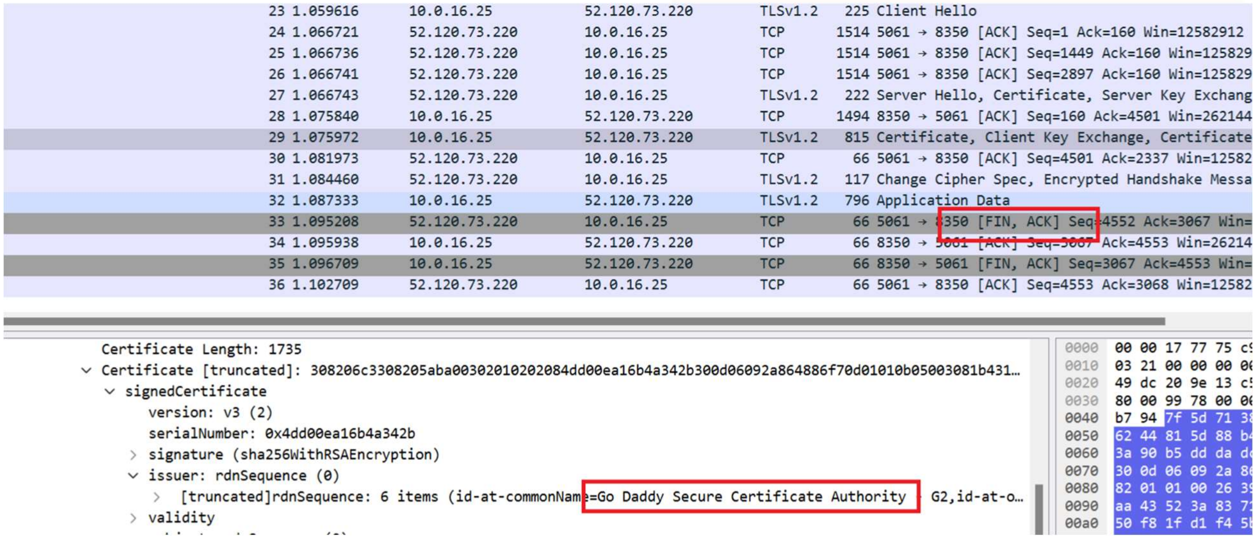
```
0000  02 00 17 02 66 8d 00
0010  00 cf f6 9c 40 00 76
0020  10 19 13 c5 20 28 aa
0030  40 01 4d dd 00 00 01
0040  ca 63 ca 3b 45 f4 34
0050  a0 28 87 d4 e5 54 06
0060  17 f5 63 d3 4b bd 46
0070  1a d3 48 86 f5 0f c6
0080  2c f0 9d a9 3a 8e ad
0090  db d0 48 48 2f 1b 76
00a0  a4 89 2b 91 eb 95 20
00b0  d1 d4 1d 8b ed 0d 00
00c0  04 08 05 08 06 04 01
00d0  03 02 02 06 01 06 03
```

In this exchange TLS handshake goes further. SBC sends “client hello”, as a server MSFT answers with “server hello” “certificates”. Straight upon receiving remote certificate chain SBC terminates TCP connection sending FIN. SBC was not able to verify remote certificate chain trust. From the server certificate message is obvious who signed their certificate and this issue gets fixed by adding again

"DigiCert Global Root G2" back to "trusted-ca-certificate" list of corresponding TLS profile. It may be very well this certificate was not even loaded to SBC so this has to be done as step one. Log.atcpd will reflect this situation as:

```
Jul  9 14:32:28.538 [SERVICE] (2) TLS Handshake: client >>> TLS 1.2 Alert[length 0002], fatal unknown_ca

Jul  9 14:32:28.538 [SERVICE] (2) <tlsengine.cpp:1549> SSL3 alert write:fatal:unknown CA

Jul  9 14:32:28.539 [SERVICE] (2) <tlsengine.cpp:1567> SSL_connect:error in error

Jul  9 14:32:28.539 [SERVICE] (2) <tlsengine.cpp:4237> TLSEngine::TLSMachineDOControl, appData_m = 0, n = -1

Jul  9 14:32:28.539 [SERVICE] (2) <tlsengine.cpp:5018> encrypted packet sent:

Jul  9 14:32:28.539 [SERVICE] (2) 15 03 03 00 02 02 30 length: 7

Jul  9 14:32:28.539 [ATCP] (2) <AtcpSocket.cpp:894> virtual int AtcpSocket::Send(const void*, size_t) bytes to send=7
fd=1071279

Jul  9 14:32:28.539 [ATCP] (2) <clog.cpp:98> atcpGetControlMblk: ALLOCED mBlk at 0x2f2769d0

Jul  9 14:32:28.539 [ATCP] (2) <clog.cpp:98>          (mData:0x2dbb2828,mFlags=0,mNext:(nil),len=23)

Jul  9 14:32:28.539 [ATCP] (2) <AtcpSocket.cpp:878> int AtcpSocket::sendOnePacket(mBlk*, int, const void*, int) add crsId=0 to
acme header for fd=1071279

Jul  9 14:32:28.539 [ATCP] (2) <AtcpServicePipe.cpp:1400> Asock::Send phy,vlan=0,0 cookie=0x0x171

Jul  9 14:32:28.539 [ATCP] (2) <AtcpServicePipe.cpp:1757> virtual int AtcpServicePipe::TransmitData(const void*, uint32_t)
putting on transport queue, cookie=0x0x171

Jul  9 14:32:28.539 [SERVICE] (2) <Commands.h:410> add command AtcpDataCommand 0x8539c790(2) on Transport queue # 2

Jul  9 14:32:28.539 [SERVICE] (2) <tlsengine.cpp:5047> TLSEngine::FlushNetworkBIO nFD:-1, fromBIO:7, numWrite:7, writePos:7

Jul  9 14:32:28.539 [MINOR] (2) SSL_accept failed, fatal alert sent
```

In next simulation TLS handshake will go even more further, here I'm simulating SBC sending incomplete chain with expectation that MSFT will terminate TLS handshake. To simulate this I will remove one intermediate (that signed the local cert) from "trusted-ca-certificate" list.



```
23 1.059616   10.0.16.25      52.120.73.220   TLSv1.2   225 Client Hello
24 1.066721   52.120.73.220   10.0.16.25      TCP      1514 5061 → 8350 [ACK] Seq=1 Ack=160 Win=12582912
25 1.066736   52.120.73.220   10.0.16.25      TCP      1514 5061 → 8350 [ACK] Seq=1449 Ack=160 Win=125829
26 1.066741   52.120.73.220   10.0.16.25      TCP      1514 5061 → 8350 [ACK] Seq=2897 Ack=160 Win=125829
27 1.066743   52.120.73.220   10.0.16.25      TLSv1.2   222 Server Hello, Certificate, Server Key Exchang
28 1.075840   10.0.16.25      52.120.73.220   TCP      1494 8350 → 5061 [ACK] Seq=160 Ack=4501 Win=262144
29 1.075972   10.0.16.25      52.120.73.220   TLSv1.2   815 Certificate, Client Key Exchange, Certificate
30 1.081973   52.120.73.220   10.0.16.25      TCP        66 5061 → 8350 [ACK] Seq=4501 Ack=2337 Win=12582
31 1.084460   52.120.73.220   10.0.16.25      TLSv1.2   117 Change Cipher Spec, Encrypted Handshake Messa
32 1.087333   10.0.16.25      52.120.73.220   TLSv1.2   796 Application Data
33 1.095208   52.120.73.220   10.0.16.25      TCP        66 5061 → 8350 [FIN, ACK] Seq=4552 Ack=3067 Win=
34 1.095938   10.0.16.25      52.120.73.220   TCP        66 8350 → 5061 [ACK] Seq=3067 Ack=4553 Win=26214
35 1.096709   10.0.16.25      52.120.73.220   TCP        66 8350 → 5061 [FIN, ACK] Seq=3067 Ack=4553 Win=
36 1.102709   52.120.73.220   10.0.16.25      TCP        66 5061 → 8350 [ACK] Seq=4553 Ack=3068 Win=12582
```

```
    Certificate Length: 1735
  ⌄ Certificate [truncated]: 308206c3308205aba00302010202084dd00ea16b4a342b300d06092a864886f70d01010b05003081b431…
    ⌄ signedCertificate
        version: v3 (2)
        serialNumber: 0x4dd00ea16b4a342b
      › signature (sha256WithRSAEncryption)
      ⌄ issuer: rdnSequence (0)
        › [truncated]rdnSequence: 6 items (id-at-commonName=Go Daddy Secure Certificate Authority - G2,id-at-o…
      › validity
```

```
0000  00 00 17 77 75 c
0010  03 21 00 00 00 0
0020  49 dc 20 9e 13 c
0030  80 00 99 78 00 0
0040  b7 94 7f 5d 71 3
0050  62 44 81 5d 88 b
0060  3a 90 b5 dd da d
0070  30 0d 06 09 2a 8
0080  82 01 01 00 26 3
0090  aa 43 52 3a 83 7
00a0  50 f8 1f d1 f4 5
```

In a screenshot above TLS handshake goes further and SBC as client verifies server side certificate successfully. Upon server request it also sends its certificate in frame 29. However as it sends incomplete data, only signed cert with no intermediates server will consider it incomplete and terminate TCP connection with FIN. Action point here is to verify that SBC side full chain is loaded to SBC.

SRTP application layer negotiation failure happens post TLS is up and usually reflects as human readable session termination with SIP 488 “Not acceptable here” replied back to sender’s SIP INVITE. Both parties should agree on supported ciphers and sdes-profile should be tuned accordingly.

Abnormal TLS cases

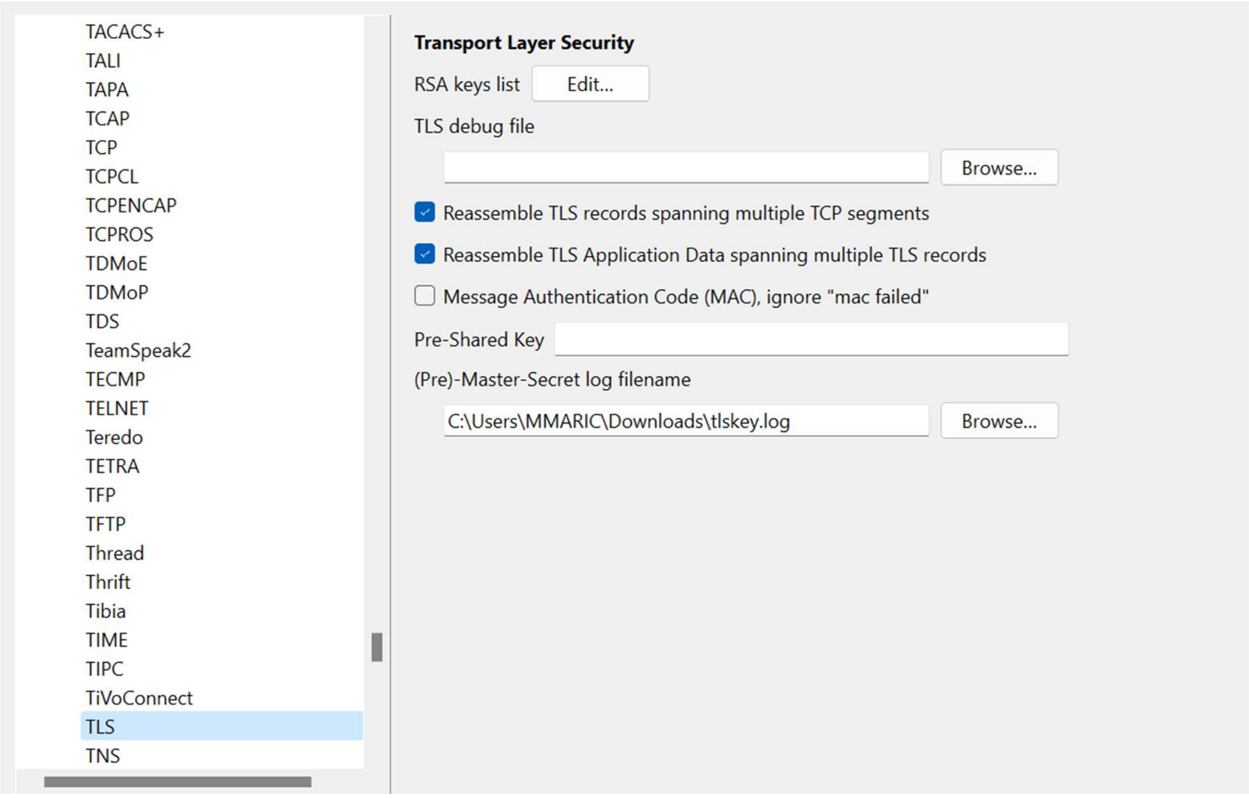
It may happen for whatever reasons that application layer logs in SBC cannot be checked and there is no OCOM in place while we have healthy indication that underlying TLS and SRTP are all fine. With recent 9.2 feature SBC may log TLSv1.2 and TLSv1.3 pre-master and master secret for a TLS connection that helps decrypting traces in wireshark.

In my next example I have healthy TLS session-agent indication but my calls are failing. There is no sipmsg.log available nor OCOM in place. An option to go with is following:

- Setup packet-trace local on desired network-interface
- Configure system-config parameter log-tls-key

- Attempt the failing call and collect the pcap from /opt/traces(note that pcap may be from elsewhere in network)
- Fetch the pre-master and master key from /opt/logs. File is called tlskey.log
- Attempt to decrypt messages in wireshark

In wireshark one needs to go TLS protocol preferences and target the file fetched from SBC as looks below:



Transport Layer Security

RSA keys list

TLS debug file

☑ Reassemble TLS records spanning multiple TCP segments

☑ Reassemble TLS Application Data spanning multiple TLS records

☐ Message Authentication Code (MAC), ignore "mac failed"

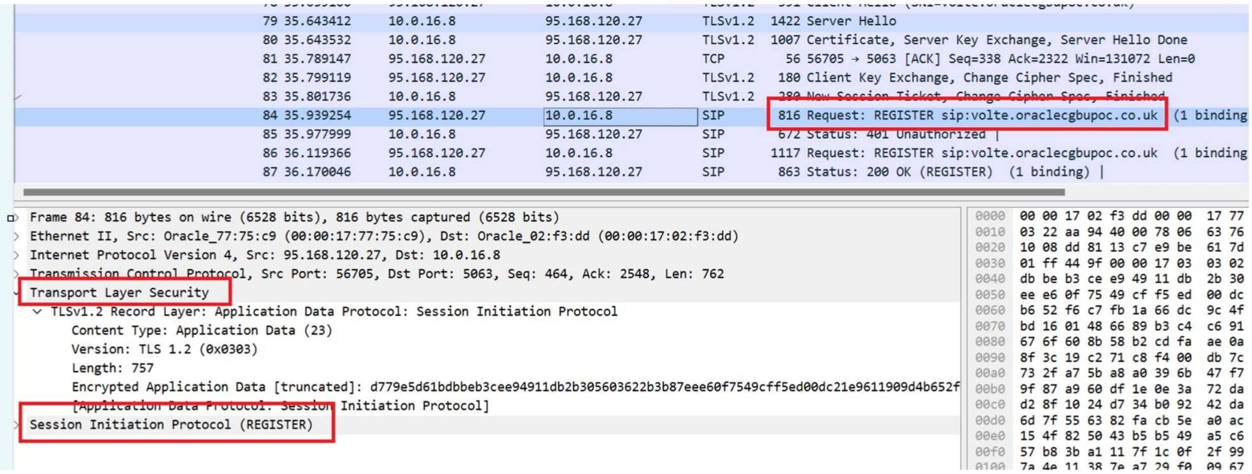
Pre-Shared Key

(Pre)-Master-Secret log filename

C:\Users\MMARIC\Downloads\tlskey.log

Only step remaining is a right click on TLS packet under inspection and use “decode as” choosing SIP given packets may be reassembled.

TLS packets became visible as raw data:

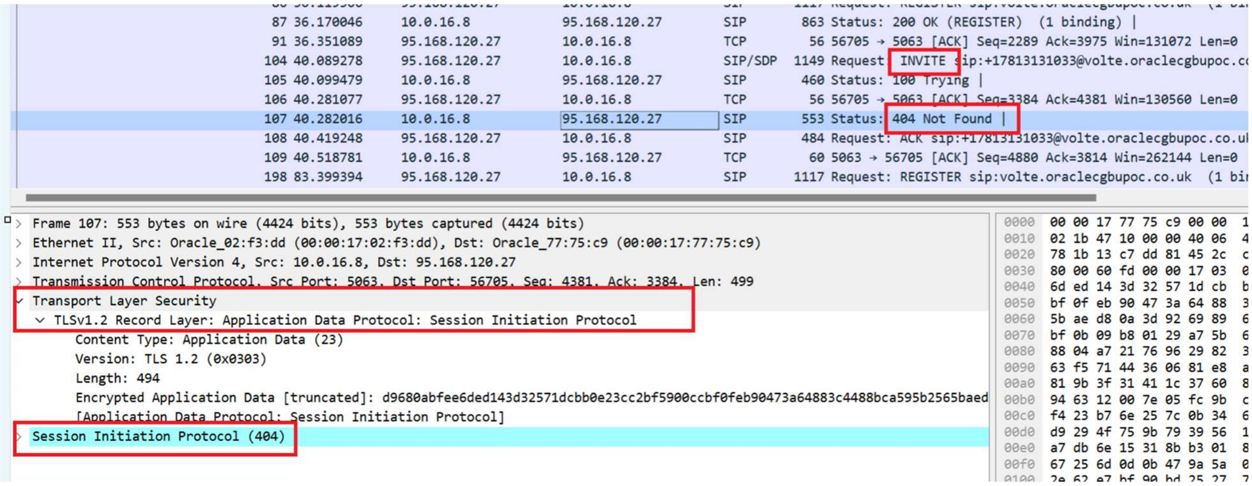


```
 79 35.643412   10.0.16.8       95.168.120.27   TLSv1.2  1422 Server Hello
 80 35.643532   10.0.16.8       95.168.120.27   TLSv1.2  1007 Certificate, Server Key Exchange, Server Hello Done
 81 35.789147   95.168.120.27   10.0.16.8       TCP        56 56705 → 5063 [ACK] Seq=338 Ack=2322 Win=131072 Len=0
 82 35.799119   95.168.120.27   10.0.16.8       TLSv1.2   180 Client Key Exchange, Change Cipher Spec, Finished
 83 35.801736   10.0.16.8       95.168.120.27   TLSv1.2   280 New Session Ticket, Change Cipher Spec, Finished
 84 35.939254   95.168.120.27   10.0.16.8       SIP       816 Request: REGISTER sip:volte.oraclecgbupoc.co.uk  (1 binding
 85 35.977999   10.0.16.8       95.168.120.27   SIP       672 Status: 401 Unauthorized |
 86 36.119366   95.168.120.27   10.0.16.8       SIP      1117 Request: REGISTER sip:volte.oraclecgbupoc.co.uk  (1 binding
 87 36.170046   10.0.16.8       95.168.120.27   SIP       863 Status: 200 OK (REGISTER)  (1 binding) |
```

```
Frame 84: 816 bytes on wire (6528 bits), 816 bytes captured (6528 bits)
Ethernet II, Src: Oracle_77:75:c9 (00:00:17:77:75:c9), Dst: Oracle_02:f3:dd (00:00:17:02:f3:dd)
Internet Protocol Version 4, Src: 95.168.120.27, Dst: 10.0.16.8
Transmission Control Protocol, Src Port: 56705, Dst Port: 5063, Seq: 464, Ack: 2548, Len: 762
Transport Layer Security
  TLSv1.2 Record Layer: Application Data Protocol: Session Initiation Protocol
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 757
      Encrypted Application Data [truncated]: d779e5d61bdbbeb3cee94911db2b305603622b3b87eee60f7549cff5ed00dc21e9611909d4b652f
      [Application Data Protocol: Session Initiation Protocol]
Session Initiation Protocol (REGISTER)
```

```
0000  00 00 17 02 f3 dd 00 00  17 77
0010  03 22 aa 94 40 00 78 06  63 76
0020  10 08 dd 81 13 c7 e9 be  61 7d
0030  01 ff 44 9f 00 00 17 03  03 02
0040  db be b3 ce e9 49 11 db  2b 30
0050  ee e6 0f 75 49 cf f5 ed  00 dc
0060  b6 52 f6 c7 fb 1a 66 dc  9c 4f
0070  bd 16 01 48 66 89 b3 c4  c6 91
0080  67 6f 60 8b 58 b2 cd fa  ae 0a
0090  8f 3c 19 c2 71 c8 f4 00  db 7c
00a0  73 2f a7 5b a8 a0 39 6b  47 f7
00b0  9f 87 a9 60 df 1e 0e 3a  72 da
00c0  d2 8f 10 24 d7 34 b0 92  42 da
00d0  6d 7f 55 63 82 fa cb 5e  a0 ac
00e0  15 4f 82 50 43 b5 b5 49  a5 c6
00f0  57 b8 3b a1 11 7f 1c 0f  2f 99
```

At point we'd normally see just encrypted application data now we have a full view over decrypted content. Furthermore reason of the call failure can be further explored:



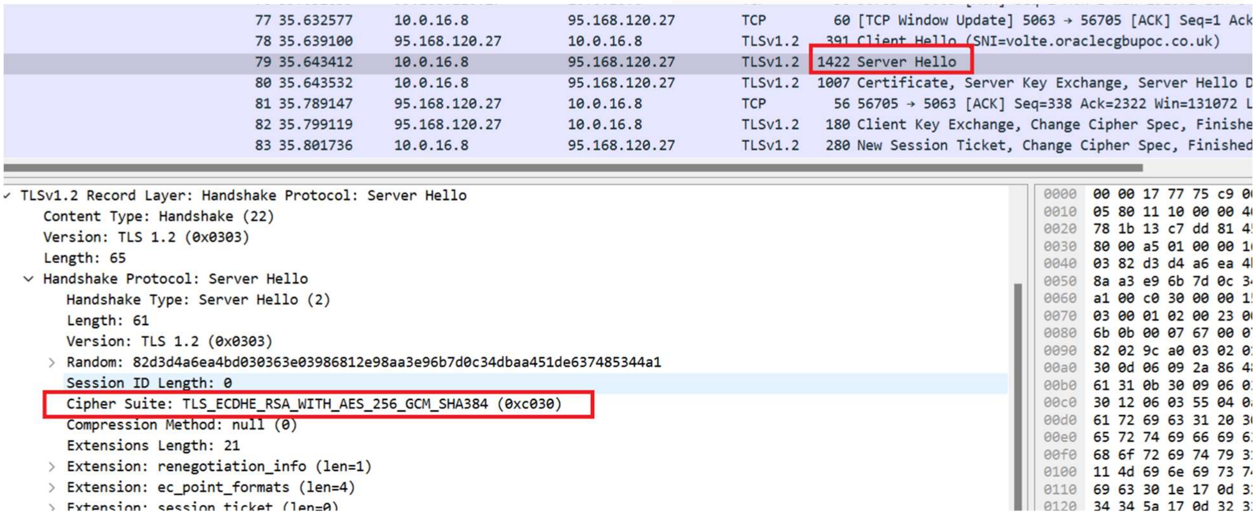
```
 87 36.170046   10.0.16.8       95.168.120.27   SIP       863 Status: 200 OK (REGISTER)  (1 binding) |
 91 36.351089   95.168.120.27   10.0.16.8       TCP        56 56705 → 5063 [ACK] Seq=2289 Ack=3975 Win=131072 Len=0
104 40.089278   95.168.120.27   10.0.16.8       SIP/SDP  1149 Request: INVITE sip:+17813131033@volte.oraclecgbupoc.c
105 40.099479   10.0.16.8       95.168.120.27   SIP       460 Status: 100 Trying |
106 40.281077   95.168.120.27   10.0.16.8       TCP        56 56705 → 5063 [ACK] Seq=3384 Ack=4381 Win=130560 Len=0
107 40.282016   10.0.16.8       95.168.120.27   SIP       553 Status: 404 Not Found |
108 40.419248   95.168.120.27   10.0.16.8       SIP       484 Request: ACK sip:+17813131033@volte.oraclecgbupoc.co.u
109 40.518781   10.0.16.8       95.168.120.27   TCP        60 5063 → 56705 [ACK] Seq=4880 Ack=3814 Win=262144 Len=0
198 83.399394   95.168.120.27   10.0.16.8       SIP      1117 Request: REGISTER sip:volte.oraclecgbupoc.co.uk  (1 bi
```

```
Frame 107: 553 bytes on wire (4424 bits), 553 bytes captured (4424 bits)
Ethernet II, Src: Oracle_02:f3:dd (00:00:17:02:f3:dd), Dst: Oracle_77:75:c9 (00:00:17:77:75:c9)
Internet Protocol Version 4, Src: 10.0.16.8, Dst: 95.168.120.27
Transmission Control Protocol, Src Port: 5063, Dst Port: 56705, Seq: 4381, Ack: 3384, Len: 499
Transport Layer Security
  TLSv1.2 Record Layer: Application Data Protocol: Session Initiation Protocol
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 494
      Encrypted Application Data [truncated]: d9680abfee6ded143d32571dcbb0e23cc2bf5900ccbf0feb90473a64883c4488bca595b2565baed
      [Application Data Protocol: Session Initiation Protocol]
Session Initiation Protocol (404)
```

```
0000  00 00 17 77 75 c9 00 00  1
0010  02 1b 47 10 00 00 40 06  4
0020  78 1b 13 c7 dd 81 45 2c  c
0030  80 00 60 fd 00 00 17 03  0
0040  6d ed 14 3d 32 57 1d cb  b
0050  bf 0f eb 90 47 3a 64 88  3
0060  5b ae d8 0a 3d 92 69 89  6
0070  bf 0b 09 b8 01 29 a7 5b  6
0080  88 04 a7 21 76 96 29 82  3
0090  63 f5 71 44 36 06 81 e8  a
00a0  81 9b 3f 31 41 1c 37 60  8
00b0  94 63 12 00 7e 05 fc 9b  c
00c0  f4 23 b7 6e 25 7c 0b 34  6
00d0  d9 29 4f 75 9b 79 39 56  1
00e0  a7 db 6e 15 31 8b b3 01  8
00f0  67 25 6d 0d 0b 47 9a 5a  0
```

As an outcome, RC of the call failure, seems to be in routing and SBC may be checked for the proper call routing configuration fine tuning.

For sake of completeness worth exposing for the case above what the TLS cipher suit negotiated was:



```
77 35.632577   10.0.16.8      95.168.120.27   TCP       60 [TCP Window Update] 5063 → 56705 [ACK] Seq=1 Ack
78 35.639100   95.168.120.27  10.0.16.8       TLSv1.2  391 Client Hello (SNI=volte.oraclecgbupoc.co.uk)
79 35.643412   10.0.16.8      95.168.120.27   TLSv1.2 1422 Server Hello
80 35.643532   10.0.16.8      95.168.120.27   TLSv1.2 1007 Certificate, Server Key Exchange, Server Hello D
81 35.789147   95.168.120.27  10.0.16.8       TCP       56 56705 → 5063 [ACK] Seq=338 Ack=2322 Win=131072 L
82 35.799119   95.168.120.27  10.0.16.8       TLSv1.2  180 Client Key Exchange, Change Cipher Spec, Finishe
83 35.801736   10.0.16.8      95.168.120.27   TLSv1.2  280 New Session Ticket, Change Cipher Spec, Finished

✓ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 65
  ∨ Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 61
      Version: TLS 1.2 (0x0303)
    > Random: 82d3d4a6ea4bd030363e03986812e98aa3e96b7d0c34dbaa451de637485344a1
      Session ID Length: 0
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
      Compression Method: null (0)
      Extensions Length: 21
    > Extension: renegotiation_info (len=1)
    > Extension: ec_point_formats (len=4)
    > Extension: session_ticket (len=0)

0000  00 00 17 77 75 c9
0010  05 80 11 10 00 00
0020  78 1b 13 c7 dd 81
0030  80 00 a5 01 00 00
0040  03 82 d3 d4 a6 ea
0050  8a a3 e9 6b 7d 0c
0060  a1 00 c0 30 00 00
0070  03 00 01 02 00 23
0080  6b 0b 00 07 67 00
0090  82 02 9c a0 03 02
00a0  30 0d 06 09 2a 86
00b0  61 31 0b 30 09 06
00c0  30 12 06 03 55 04
00d0  61 72 69 63 31 20
00e0  65 72 74 69 66 69
00f0  68 6f 72 69 74 79
0100  11 4d 69 6e 69 73
0110  69 63 30 1e 17 0d
0120  34 34 5a 17 0d 32
```

Abnormal SRTP cases

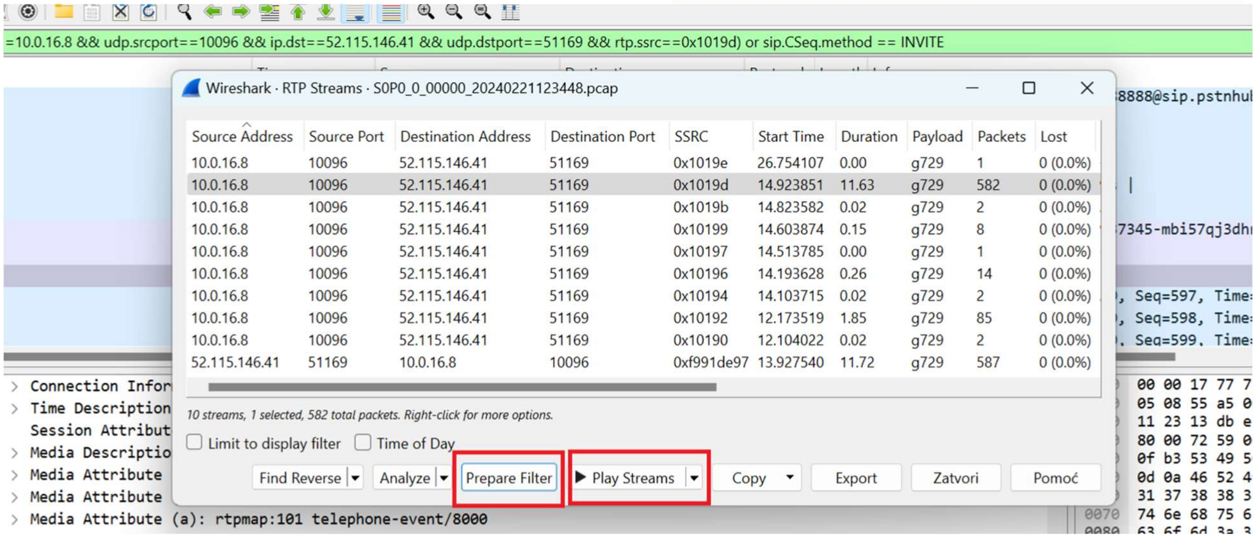
There is no similar equivalent in SBC or in wireshark that would allow us to decrypt SRTP easy. However, below an example how SRTP stream fetched on network level may be decrypted with open source tool srtp-decrypt.

GitHub - gteissier/srtp-decrypt: Deciphers SRTP packets

Project was compiled on testing Oracle linux machine.

Attempting to decrypt SRTP assumption is that we had successful TLS handshake and that we have a view over a SIP call in clear to grab crypto key, also we need network layer of SRTP capture itself(.pcap). It may be challenging to isolate proper RTP stream from wireshark but once isolated single stream direction has to be saved as an input for srtp-decrypt application. Procedure looks as below:

Isolating proper RTP stream, filtering and saving to file:



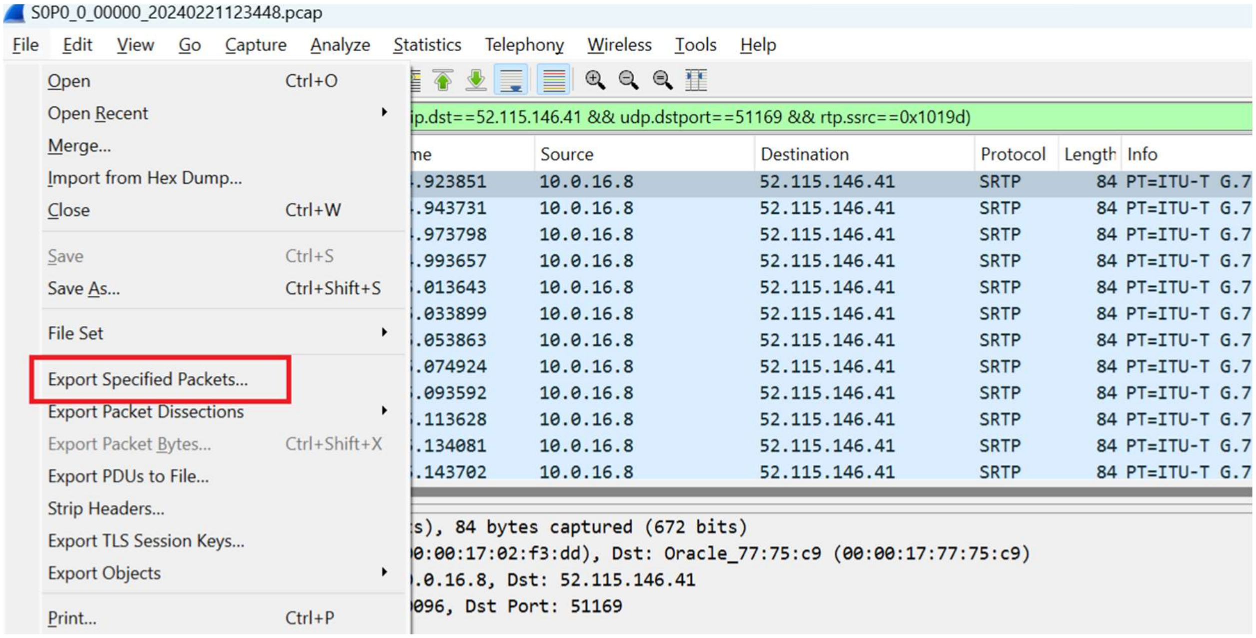
`=10.0.16.8 && udp.srcport==10096 && ip.dst==52.115.146.41 && udp.dstport==51169 && rtp.ssrc==0x1019d) or sip.CSeq.method == INVITE`

Wireshark · RTP Streams · S0P0_0_00000_20240221123448.pcap

Source Address	Source Port	Destination Address	Destination Port	SSRC	Start Time	Duration	Payload	Packets	Lost
10.0.16.8	10096	52.115.146.41	51169	0x1019e	26.754107	0.00	g729	1	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x1019d	14.923851	11.63	g729	582	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x1019b	14.823582	0.02	g729	2	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10199	14.603874	0.15	g729	8	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10197	14.513785	0.00	g729	1	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10196	14.193628	0.26	g729	14	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10194	14.103715	0.02	g729	2	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10192	12.173519	1.85	g729	85	0 (0.0%)
10.0.16.8	10096	52.115.146.41	51169	0x10190	12.104022	0.02	g729	2	0 (0.0%)
52.115.146.41	51169	10.0.16.8	10096	0xf991de97	13.927540	11.72	g729	587	0 (0.0%)

10 streams, 1 selected, 582 total packets. Right-click for more options.

One may verify first that trying to play streams we hear only crackling noise. Afterwards hitting “prepare filter” wireshark will filter the single RTP stream direction for which single crypto applies. Such file has to be saved as below



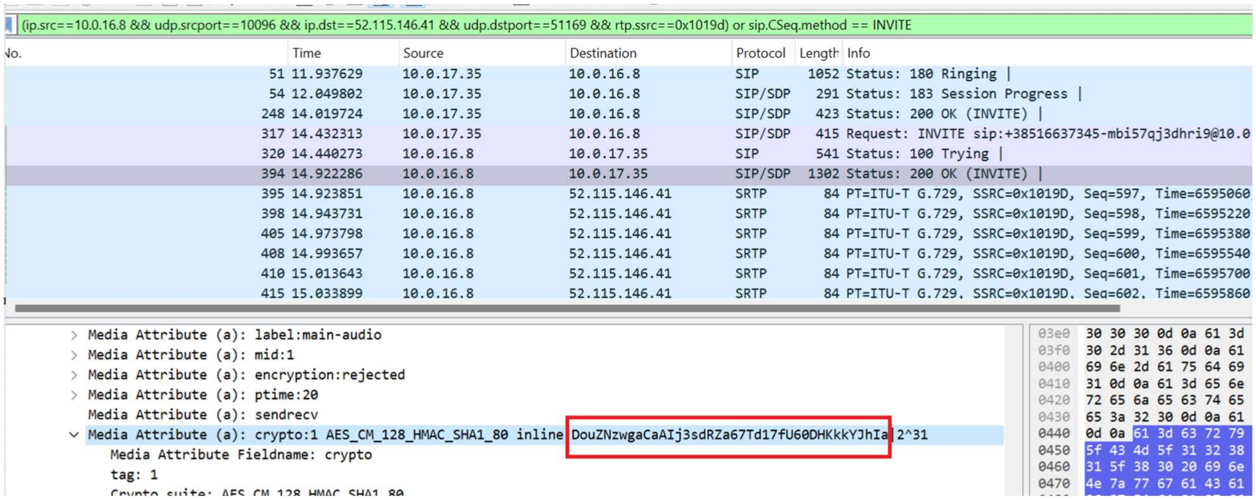
S0P0_0_00000_20240221123448.pcap

`ip.dst==52.115.146.41 && udp.dstport==51169 && rtp.ssrc==0x1019d)`

me	Source	Destination	Protocol	Length	Info
.923851	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.943731	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.973798	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.993657	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.013643	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.033899	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.053863	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.074924	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.093592	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.113628	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.134081	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7
.143702	10.0.16.8	52.115.146.41	SRTP	84	PT=ITU-T G.7

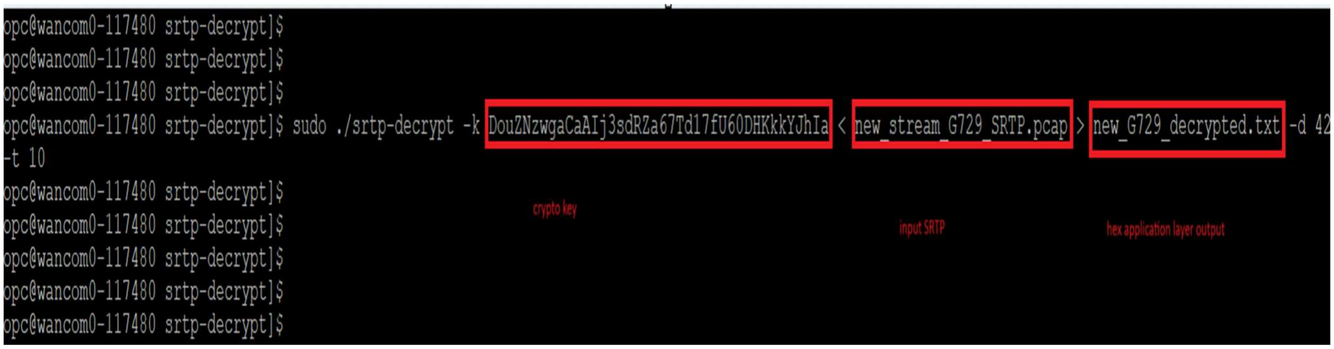
```
s), 84 bytes captured (672 bits)
0:00:17:02:f3:dd), Dst: Oracle_77:75:c9 (00:00:17:77:75:c9)
.0.16.8, Dst: 52.115.146.41
096, Dst Port: 51169
```

Once we have a filtered .pcap remaining is to grab 40bytes BASE64 crypto string that consists of master key and salt.



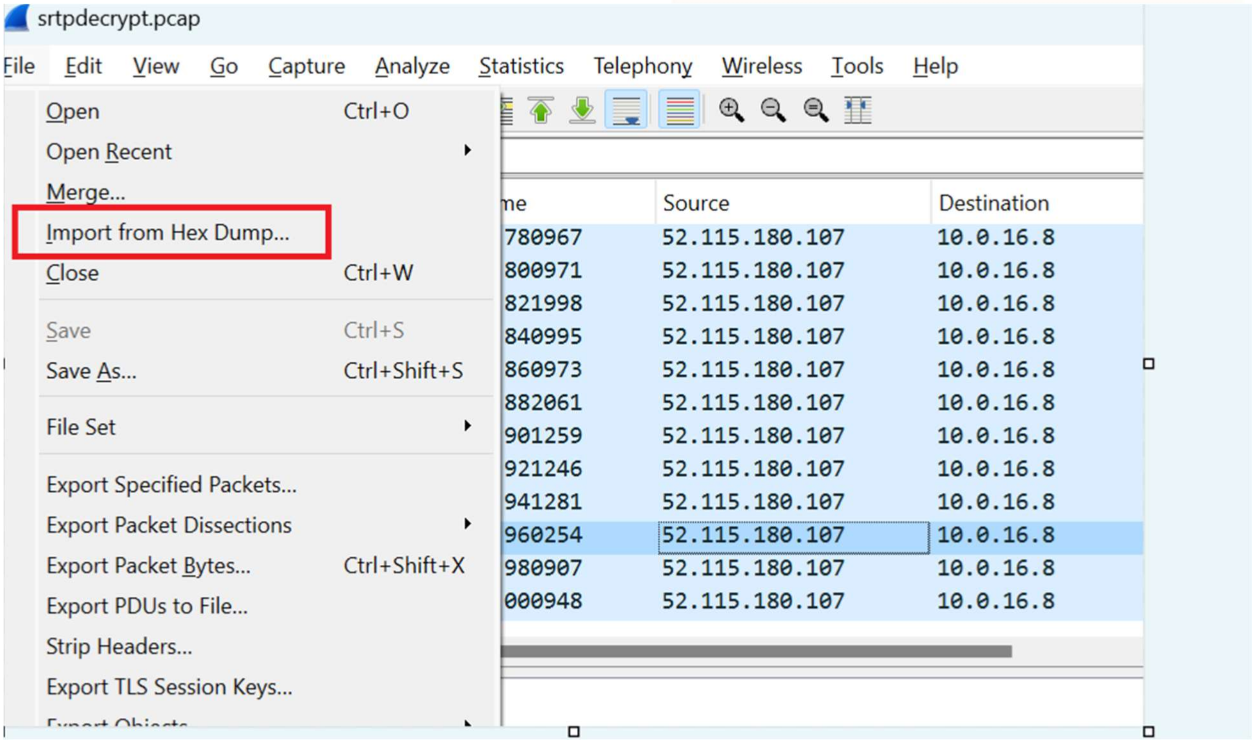
```
(ip.src==10.0.16.8 && udp.srcport==10096 && ip.dst==52.115.146.41 && udp.dstport==51169 && rtp.ssrc==0x1019d) or sip.CSeq.method == INVITE

No.    Time        Source        Destination     Protocol  Length Info
    51 11.937629   10.0.17.35    10.0.16.8       SIP       1052 Status: 180 Ringing |
    54 12.049802   10.0.17.35    10.0.16.8       SIP/SDP    291 Status: 183 Session Progress |
   248 14.019724   10.0.17.35    10.0.16.8       SIP/SDP    423 Status: 200 OK (INVITE) |
   317 14.432313   10.0.17.35    10.0.16.8       SIP/SDP    415 Request: INVITE sip:+38516637345-mbi57qj3dhri9@10.0
   320 14.440273   10.0.16.8     10.0.17.35      SIP        541 Status: 100 Trying |
   394 14.922286   10.0.16.8     10.0.17.35      SIP/SDP   1302 Status: 200 OK (INVITE) |
   395 14.923851   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=597, Time=6595060
   398 14.943731   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=598, Time=6595220
   405 14.973798   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=599, Time=6595380
   408 14.993657   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=600, Time=6595540
   410 15.013643   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=601, Time=6595700
   415 15.033899   10.0.16.8     52.115.146.41   SRTP        84 PT=ITU-T G.729, SSRC=0x1019D, Seq=602, Time=6595860

    > Media Attribute (a): label:main-audio
    > Media Attribute (a): mid:1
    > Media Attribute (a): encryption:rejected
    > Media Attribute (a): ptime:20
      Media Attribute (a): sendrecv
    v Media Attribute (a): crypto:1 AES_CM_128_HMAC_SHA1_80 inline DouZNzwgaCaAIj3sdRZa67Td17fU60DHKkkYJhIa|2^31
        Media Attribute Fieldname: crypto
        tag: 1
        Crypto suite: AES_CM_128_HMAC_SHA1_80

03e0  30 30 30 0d 0a 61 3d
03f0  30 2d 31 36 0d 0a 61
0400  69 6e 2d 61 75 64 69
0410  31 0d 0a 61 3d 65 6e
0420  72 65 6a 65 63 74 65
0430  65 3a 32 30 0d 0a 61
0440  0d 0a 61 3d 63 72 79
0450  5f 43 4d 5f 31 32 38
0460  31 5f 38 30 20 69 6e
0470  4e 7a 77 67 61 43 61
```

As of that point we are ready for decryption. Available options within a tool are total frame offset before the payload and srtp authentication tag length(defaults are assumed that equal 42bytes and 10bytes for latter mentioned srtp auth tag).

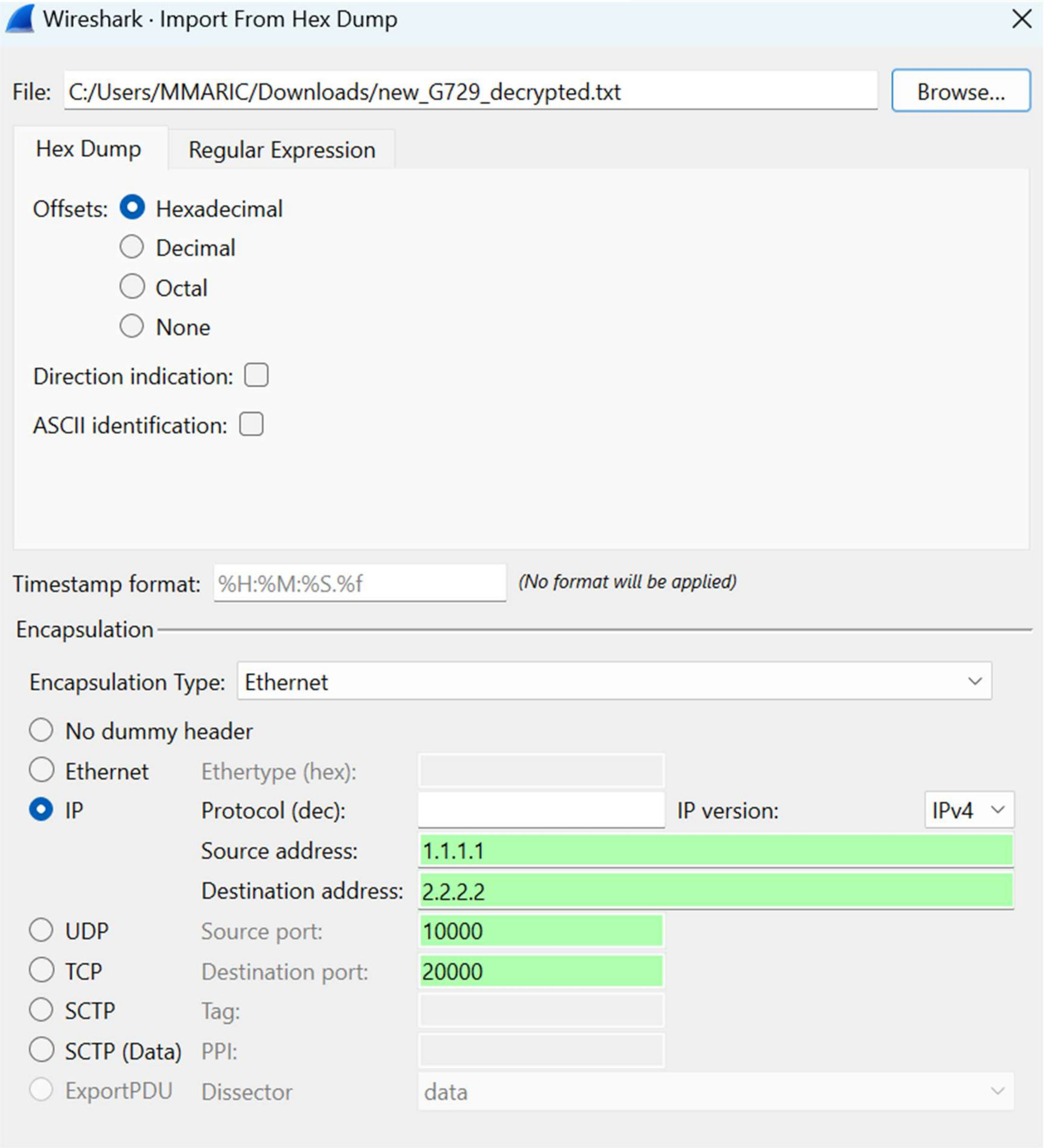


```
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$ sudo ./srtp-decrypt -k DouZNzwgaCaAIj3sdRZa67Td17fU60DHKkkYJhIa < new_stream_G729_SRTP.pcap > new_G729_decrypted.txt -d 42
-t 10
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
opc@wancom0-117480 srtp-decrypt]$
```

Application takes SRTP .pcap as an input and provides application level decrypted hex stream (only RTP content without lower layers). Output form is not an issue as hex dump can be easily imported to wireshark adding fake lower layers:



me	Source	Destination
780967	52.115.180.107	10.0.16.8
800971	52.115.180.107	10.0.16.8
821998	52.115.180.107	10.0.16.8
840995	52.115.180.107	10.0.16.8
860973	52.115.180.107	10.0.16.8
882061	52.115.180.107	10.0.16.8
901259	52.115.180.107	10.0.16.8
921246	52.115.180.107	10.0.16.8
941281	52.115.180.107	10.0.16.8
960254	52.115.180.107	10.0.16.8
980907	52.115.180.107	10.0.16.8
000948	52.115.180.107	10.0.16.8



Wireshark · Import From Hex Dump

File: C:/Users/MMARIC/Downloads/new_G729_decrypted.txt

Offsets:
● Hexadecimal
○ Decimal
○ Octal
○ None

Direction indication: ☐

ASCII identification: ☐

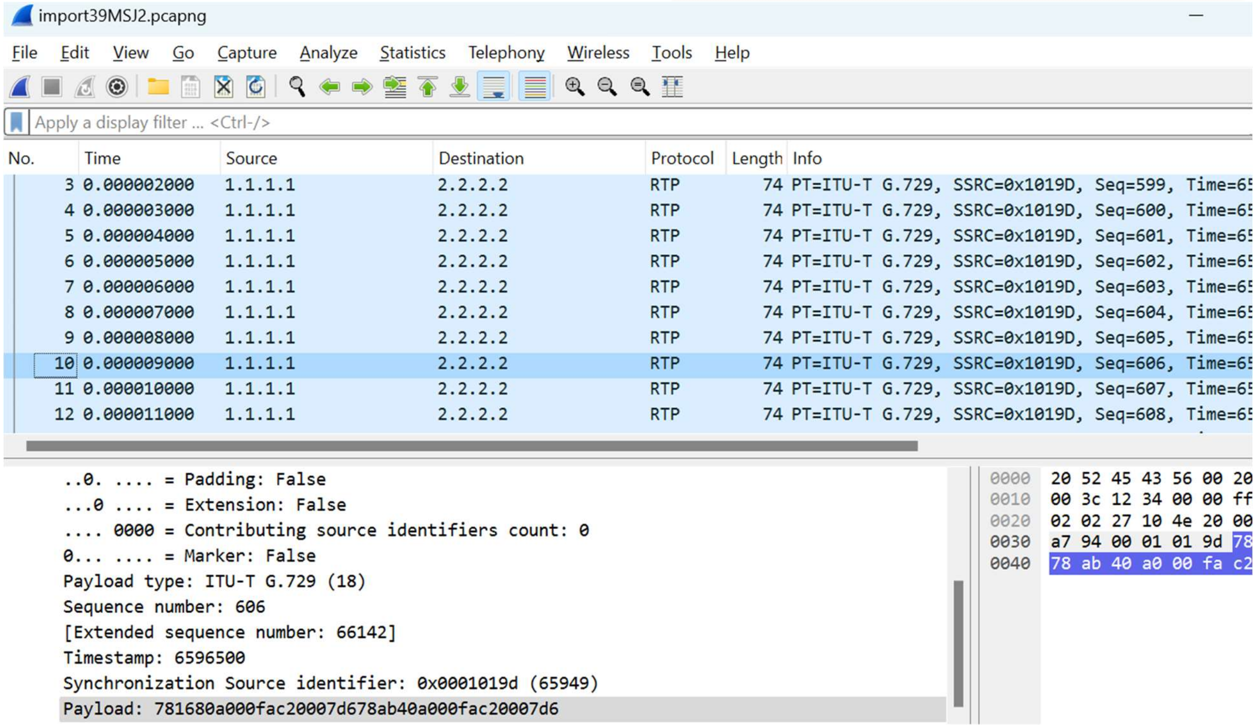
Timestamp format: %H:%M:%S.%f *(No format will be applied)*

Encapsulation

Encapsulation Type: Ethernet

○ No dummy header
○ Ethernet Ethertype (hex):
● IP Protocol (dec): IP version: IPv4
Source address: 1.1.1.1
Destination address: 2.2.2.2
○ UDP Source port: 10000
○ TCP Destination port: 20000
○ SCTP Tag:
○ SCTP (Data) PPI:
○ ExportPDU Dissector data

By putting a thick on IP header and UDP we insert fake destination/source IP and port. Hitting import wireshark displays our decrypted SRTP stream

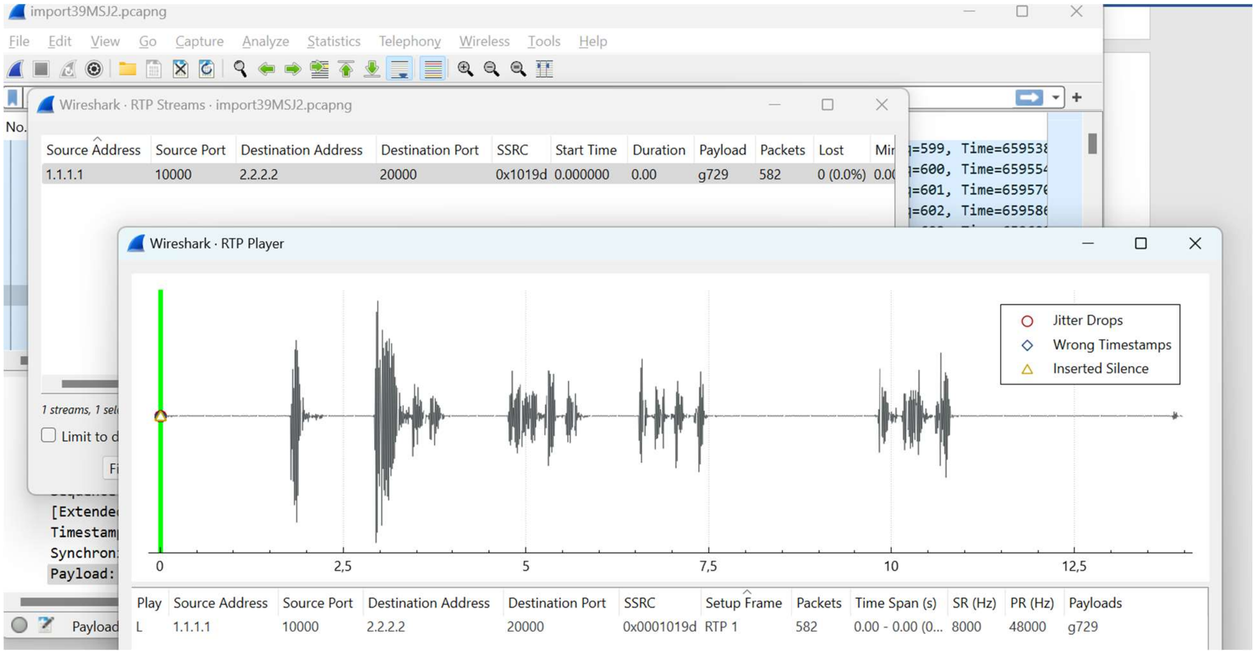


```
No.   Time          Source     Destination   Protocol  Length  Info
    3 0.000002000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=599, Time=65
    4 0.000003000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=600, Time=65
    5 0.000004000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=601, Time=65
    6 0.000005000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=602, Time=65
    7 0.000006000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=603, Time=65
    8 0.000007000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=604, Time=65
    9 0.000008000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=605, Time=65
   10 0.000009000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=606, Time=65
   11 0.000010000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=607, Time=65
   12 0.000011000   1.1.1.1    2.2.2.2       RTP           74  PT=ITU-T G.729, SSRC=0x1019D, Seq=608, Time=65
```

```
..0. .... = Padding: False
...0 .... = Extension: False
.... 0000 = Contributing source identifiers count: 0
0... .... = Marker: False
Payload type: ITU-T G.729 (18)
Sequence number: 606
[Extended sequence number: 66142]
Timestamp: 6596500
Synchronization Source identifier: 0x0001019d (65949)
Payload: 781680a000fac20007d678ab40a000fac20007d6
```

```
0000  20 52 45 43 56 00 20
0010  00 3c 12 34 00 00 ff
0020  02 02 27 10 4e 20 00
0030  a7 94 00 01 01 9d 78
0040  78 ab 40 a0 00 fa c2
```

Attempting to play the stream this time it will audible in G729



Source Address	Source Port	Destination Address	Destination Port	SSRC	Start Time	Duration	Payload	Packets	Lost	Mi
1.1.1.1	10000	2.2.2.2	20000	0x1019d	0.000000	0.00	g729	582	0 (0.0%)	0.0

Play	Source Address	Source Port	Destination Address	Destination Port	SSRC	Setup Frame	Packets	Time Span (s)	SR (Hz)	PR (Hz)	Payloads
L	1.1.1.1	10000	2.2.2.2	20000	0x0001019d	RTP 1	582	0.00 - 0.00 (0...	8000	48000	g729