Figure 1: UTP Logo

TEB2093 Computer Security - Lab 06

Members

- Ammar Farhan Bin Mohamad Rizam (22006911)
- Amisya Fareezan Binti Mohd Fadhil (22007082)
- Ahmad Anas Bin Azhar (22005996)
- Muhammad Hanis Afifi Bin Azmi (22001602)

Table of Contents

Task 1

Deriving the private key from the following public key:

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

Task 1 - Code

```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *message, const BIGNUM *number) {
    char *number_str = BN_bn2hex(number);
    printf("%s %s\n", message, number_str);
    OPENSSL_free(number_str);
}

int main(void) {
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *phi = BN_new();
    BIGNUM *d = BN_new();

    BN_CTX *ctx = BN_CTX_new();

    // values according to instructions
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    // n = p * q
    BN_mul(n, p, q, ctx);

    // phi(n) = (p - 1)(q - 1)
    BIGNUM *one = BN_new();
    BN_one(one);
    BN_sub(p, p, one);
    BN_sub(q, q, one);
    BN_mul(phi, p, q, ctx);

    // d = e^-1 mod phi(n)
    BN_mod_inverse(d, e, phi, ctx);

    printf("Public Key (e, n):\n");
    printBN("\te = ", e);
    printBN("\tn = ", n);
    printf("Private Key (e, n):\n");
    printBN("\td = ", d);
    printBN("\tn = ", n);

    BN_free(p);
    BN_free(q);
    BN_free(e);
    BN_free(n);
    BN_free(phi);
    BN_free(d);

    BN_CTX_free(ctx);

    return 0;
}
```

Task 1 - Output

```
Public Key (e, n):
    e =  0D88C3
    n =  E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
Private Key (e, n):
    d =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
    n =  E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
```

Task 2

Encrypting a message:

```
M = A top secret!
```

Using the public key:

```
e = 010001
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
```

Verify that the encryption is done correctly, by decrypting the encrypted message using the private key:

```
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
```

Task 2 - Code

```python
#!/usr/bin/env python3


class RSAPublicKey:
    def __init__(self, e: str, n: str):
        self.e = int(e, 16)
        self.n = int(n, 16)

    def __repr__(self) -> str:
        return f"Public Key:\n\te = {hex(self.e)}\n\tn = {hex(self.n)}"


class RSAPrivateKey:
    def __init__(self, d: str, n: str):
        self.d = int(d, 16)
        self.n = int(n, 16)

    def __repr__(self) -> str:
        return f"PrivateKey:\n\td = {hex(self.d)}\n\tn = {hex(self.n)}"


class RSA:
    @staticmethod
    def encrypt(message: str, public_key: RSAPublicKey) -> int:
        message_hex = message.encode("utf-8").hex()
        message_int = int(message_hex, 16)
        return pow(message_int, public_key.e, public_key.n)

    @staticmethod
    def decrypt(encrypted_message: int, private_key: RSAPrivateKey) -> str:
        message_decrypted = pow(encrypted_message, private_key.d, private_key.n)
        message_decrypted_hex = hex(message_decrypted)[2:]
        return bytes.fromhex(message_decrypted_hex).decode("utf-8", errors="ignore")
```

```python
if __name__ == "__main__":
    # public key from instructions
    public_key = RSAPublicKey(
        "010001", "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5"
    )

    # private key from instructions
    private_key = RSAPrivateKey(
        "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D",
        "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5",
    )

    print("+--------------------------------------------------------------------------+")
    print("|                               Task 02                                    |")
    print("+--------------------------------------------------------------------------+")

    message_str = "A top secret!"
    message_encrypted = RSA.encrypt(message_str, public_key)
    message_decrypted = RSA.decrypt(message_encrypted, private_key)

    print("[*] Logging...")
    print(f"Original message: {message_str}")
    print(f"\tEncrypted message: {message_encrypted}")
    print(f"\tDecrypted message: {message_decrypted}")
    print(
        "[+] Decryption successful!\n"
        if message_decrypted == message_str
        else "[-] Decryption unsuccessful.\n"
    )
```

Task 2 - Output

```
+--------------------------------------------------------------------------+
|                               Task 02                                    |
+--------------------------------------------------------------------------+
[*] Logging...
Original message: A top secret!
    Encrypted message: 50518525371929684556329211359721949099156057889496242376979402393388933577436
    Decrypted message: A top secret!
[+] Decryption successful!
```

Task 3

Decrypting a message:

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

Using the private key:

```
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
```

Task 3 - Code

```python
#!/usr/bin/env python3


class RSAPrivateKey:
    def __init__(self, d: str, n: str):
        self.d = int(d, 16)
        self.n = int(n, 16)

    def __repr__(self) -> str:
        return f"PrivateKey:\n\td = {hex(self.d)}\n\tn = {hex(self.n)}"


class RSA:
    @staticmethod
    def decrypt(encrypted_message: int, private_key: RSAPrivateKey) -> str:
        message_decrypted = pow(encrypted_message, private_key.d, private_key.n)
        message_decrypted_hex = hex(message_decrypted)[2:]
        return bytes.fromhex(message_decrypted_hex).decode("utf-8", errors="ignore")


if __name__ == "__main__":
    # private key from instructions
    private_key = RSAPrivateKey(
        "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D",
        "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5",
    )

    print("+--------------------------------------------------------------------------+")
    print("|                               Task 03                                    |")
    print("+--------------------------------------------------------------------------+")

    message_hex = "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F"
    message_int = int(message_hex, 16)

    print("[*] Logging...")
    print(f"Encrypted message: {message_hex}")
    print(f"Decrypted message: {RSA.decrypt(message_int, private_key)}")
```

Task 3 - Output

```
+--------------------------------------------------------------------------+
|                               Task 03                                    |
+--------------------------------------------------------------------------+
[*] Logging...
Encrypted message: 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
Decrypted message: Password is dees
```

Task 4

Signing a message:

`M = I owe you $2000.`

Using the private key:

```
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
```

Task 4 - Code

```python
#!/usr/bin/env python3


class RSAPrivateKey:
    def __init__(self, d: str, n: str):
        self.d = int(d, 16)
        self.n = int(n, 16)

    def __repr__(self) -> str:
        return f"PrivateKey:\n\td = {hex(self.d)}\n\tn = {hex(self.n)}"


class RSA:
    @staticmethod
    def sign(message: str, private_key: RSAPrivateKey) -> int:
        # in reality, we use hash, but for this lab, it says don't use hash
        # hash = int.from_bytes(sha512(message.encode("utf-8")).digest(), byteorder="big")
        message_hex = message.encode("utf-8").hex()
        message_int = int(message_hex, 16)
        return pow(message_int, private_key.d, private_key.n)


if __name__ == "__main__":
    print("+-------------------------------------------------------------------------+")
    print("|                               Task 04                                   |")
    print("+-------------------------------------------------------------------------+")

    # private key from instructions
    private_key = RSAPrivateKey(
        "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D",
        "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5",
    )

    message_str = "I owe you $2000."
    message_signature = RSA.sign(message_str, private_key)

    print("[*] Logging...")
    print(f"Original message: {message_str}")
    print(f"\tSignature:\t{message_signature}\n")

    message_str = "I owe you $3000."
    message_signature = RSA.sign(message_str, private_key)

    print("[*] Logging...")
    print(f"Modified message: {message_str}")
    print(f"\tSignature:\t{message_signature}")
```

Task 4 - Output

```
+---------------------------------------------------------------------------+
|                               Task 04                                    |
+---------------------------------------------------------------------------+
[*] Logging...
Original message: I owe you $2000.
    Signature:      38737955862331189402498387291363292989447215164396465065684612292997465629899

[*] Logging...
Modified message: I owe you $3000.
    Signature:      85377692333201951919213855180904627562313128428164207106465609485406786574370
```

Task 4 - Explanation

RSA signature features:

- RSA signatures look random and unpredictable.
- Even a tiny change in the message produces a totally different signature.
- There is no pattern or relationship between different signatures.
- Reversing an RSA signature without the private key is infeasible.

Hence, this unpredictability makes RSA signatures secure against forgery and tampering.

Task 5

Verify a signature of message:

```
M = Launch a missile.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
```

Using the public key:

```
e = 010001
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

Task 5 - Code

```python
#!/usr/bin/env python3


class RSAPublicKey:
    def __init__(self, e: str, n: str):
        self.e = int(e, 16)
        self.n = int(n, 16)

    def __repr__(self) -> str:
        return f"Public Key:\n\te = {hex(self.e)}\n\tn = {hex(self.n)}"


class RSA:
    @staticmethod
    def verify(message: str, signature: int, public_key: RSAPublicKey) -> bool:
        # in reality, we use hash, but for this lab, it says don't use hash
        # hash = int.from_bytes(sha512(message.encode("utf-8")).digest(), byteorder="big")
        message_int = pow(signature, public_key.e, public_key.n)
        message_hex = hex(message_int)[2:]

        try:
            message_str = bytes.fromhex(message_hex).decode("utf-8")
        except UnicodeDecodeError:
            return False

        return message_str == message


if __name__ == "__main__":
    print("+-------------------------------------------------------------------------+")
    print("|                               Task 05                                   |")
    print("+-------------------------------------------------------------------------+")

    public_key = RSAPublicKey(
        "010001", "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115"
    )
    message_str = "Launch a missile."
    message_signature_hex = (
        "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F"
    )
    message_signature = int(message_signature_hex, 16)
    message_verification = RSA.verify(message_str, message_signature, public_key)

    print("[*] Logging...")
    print(f"Original message: {message_str}")
    print(f"\tSignature:\t{message_signature}")
    print(f"\tVerification:\t{message_verification}\n")

    message_str = "Launch a missile."
    message_signature_hex = (
        "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F"
    )
    message_signature = int(message_signature_hex, 16)
    message_verification = RSA.verify(message_str, message_signature, public_key)

    print("[*] Logging...")
    print(f"Original message: {message_str}")
    print(f"\tCorrupted Signature:\t{message_signature}")
    print(f"\tVerification:\t\t{message_verification}\n")
```

Task 5 - Output

```
+-------------------------------------------------------------------------+
|                               Task 05                                   |
+-------------------------------------------------------------------------+
[*] Logging...
Original message: Launch a missile.
        Signature:      45339830040223574130572214402551075218831845230048698262435226537506664513583
        Verification:   True

[*] Logging...
Original message: Launch a missile.
        Corrupted Signature:    45339830040223574130572214402551075218831845230048698262435226537506664513
        Verification:           False
```

Task 5 - Explanation

If the last byte of the signature is corrupted (changing 2F to 3F, which is just one-bit change), the verification process will completely fail. This happens because RSA signature verification relies on modular exponentiation, meaning even a tiny change in the signature results in a completely different output.

Task 6

Manually verify an X.509 certificate.

1. Download a certificate from real web server.

```
$ openssl s_client -connect www.example.org:443 -showcerts

Connecting to 23.45.176.102
CONNECTED(00000005)
depth=2 C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert Global Root G3
verify return:1
depth=1 C=US, O=DigiCert Inc, CN=DigiCert Global G3 TLS ECC SHA384 2020 CA1
verify return:1
depth=0 C=US, ST=California, L=Los Angeles, O=Internet Corporation for Assigned Names and Numbers, CN
verify return:1
---
Certificate chain
0 s:C=US, ST=California, L=Los Angeles, O=Internet Corporation for Assigned Names and Numbers, CN=*.e
i:C=US, O=DigiCert Inc, CN=DigiCert Global G3 TLS ECC SHA384 2020 CA1
a:PKEY: id-ecPublicKey, 256 (bit); sigalg: ecdsa-with-SHA384
v:NotBefore: Jan 15 00:00:00 2025 GMT; NotAfter: Jan 15 23:59:59 2026 GMT
-----BEGIN CERTIFICATE-----
MIIFnTCCBSSgAwIBAgIQByKnSbVYR2GW1VREXtvSVDAKBggqhkjOPQQDAzBZMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTMwMQYDVQQDEypEaWdp
Q2VydCBHbG9iYWwgRzMgVExTIEVDQyBTSEEzODQgMjAyMCBDQTEwHhcNMjUwMTE1
MDAwMDAwWhcNMjYwMTE1MjM1OTU5WjCBjjELMAkGA1UEBhMCVVMxEzARBgNVBAgT
CkNhbGlmb3JuaWExFDASBgNVBAcTC0xvcyBBbmdlbGVzMTwwOgYDVQQKEzNJbnRl
cm5ldCBDb3Jwb3JhdGlvbiBmb3IgQXNzaWduZWQgTmFtZXMgYW5kIE51bWJlcnMx
FjAUBgNVBAMMDSouZXhhbXBsZS5vcmcwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNC
AARvcLhq3uFMuzkqpTXG4X8Wcw413owfBJMz4JcqnNnlgNb2+2F0TaF4fVoDpf8+
arlyqMYsxSxpUH/NbTudhW/Mo4IDljCCA5IwHwYDVR0jBBgwFoAUiiPrnmvX+Tdd
+W0hOXaaoWfeEKgwHQYDVR0OBBYEFBJFomWJllXyCp7B3wWf3VnuZbpRMCUGA1Ud
EQQeMByCDSouZXhhbXBsZS5vcmeCC2V4YW1wbGUub3JnMD4GA1UdIAQ3MDUwMwYG
Z4EMAQICMCkwJwYIKwYBBQUHAgEWG2h0dHA6Ly93d3cuZGlnaWNlcnQuY29tL0NQ
UzAOBgNVHQ8BAf8EBAMCA4gwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMC
MIGfBgNVHR8EgZcwgZQwSKBGoESGQmh0dHA6Ly9jcmwzLmRpZ2ljZXJ0LmNvbS9E
aWdpQ2VydEdsb2JhbEczVExTRUNDU0hBMzg0MjAyMENBMS0yLmNybDBIoEagRIZC
aHR0cDovL2NybDQuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0R2xvYmFsRzNUTFNFQ0NT
SEEzODQyMDIwQ0ExLTIuY3JsMIGHBggrBgEFBQcBAQR7MHkwJAYIKwYBBQUHMAGG
GGh0dHA6Ly9vY3NwLmRpZ2ljZXJ0LmNvbTBRBggrBgEFBQcwAoZFaHR0cDovL2Nh
Y2VydHMuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0R2xvYmFsRzNUTFNFQ0NTSEEzODQy
MDIwQ0ExLTIuY3J0MAwGA1UdEwEB/wQCMAAwggF+BgorBgEEAdZ5AgQCBIIBbgSC
AWoBaAB3AJaXZL9VWJet90OHaDcIQnfp8DrV9qTzNm5GpD8PyqnGAAABlGd6xV4A
AAQDAEgwRgIhAO28p5oX3gxA0RJJ/2MaZ3zzMcyZggy2lwVQnqSpX5R3AiEAqUWx
+2l1xeojShV0ab+MbcPNg8bYvw1xb32sJOYuxKkAdQBkEcRspBLsp4kcogIuALyr
TygH1B41J6vq/tUDyX3N8AAAAZRnesVjAAAEAwBGMEQCIAzHUguIG8H+0JF72uTL
HatlorikPR/D3P/HRsyrF+44AiBGH0KcLNqcj2ZGEjChiiRf0jLdUrFKg6jnMIoV
FMlYFwB2AEmcm2neHXzs/DbezYdkprhbrwqHgBnRVVL76esp3fjDAAABlGd6xXgA
AAQDAEcwRQIgRESM73pynQ14OQSowDrC49oQXZut2nYQc2DYrX26VXgCIQDRBYhi
```

```
5U3bC19GT2EzfDLr38vkM5yNNYnYlY1En7+TczAKBggqhkjOPQQDAwNnADBkAjBH
fYRsXBNHMfs6MlztuZRrCRw0ERMhMSHe7Al5nSDz+cP3KHcdxkRWtf1xFksSljUC
ME1pXV7GC3Vq2Pmd0wPkRSngBR9Hm2X8srwINo/QvZ91oS7dUDseaBJ5w0b5oHmA
jw==
-----END CERTIFICATE-----
1 s:C=US, O=DigiCert Inc, CN=DigiCert Global G3 TLS ECC SHA384 2020 CA1
i:C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert Global Root G3
a:PKEY: id-ecPublicKey, 384 (bit); sigalg: ecdsa-with-SHA384
v:NotBefore: Apr 14 00:00:00 2021 GMT; NotAfter: Apr 13 23:59:59 2031 GMT
-----BEGIN CERTIFICATE-----
MIIDeTCCAv+gAwIBAgIQCwDpLU1tcx/KMFnHyx4YhjAKBggqhkjOPQQDAzBhMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3d3cu
ZGlnaWNlcnQuY29tMSAwHgYDVQQDExdEaWdpQ2VydCBHbG9iYWwgUm9vdCBHMzAe
Fw0yMTA0MTQwMDAwMDBaFw0zMTA0MTMyMzU5NTlaMFkxCzAJBgNVBAYTAlVTMRUw
EwYDVQQKEwxEaWdpQ2VydCBJbmMxMzAxBgNVBAMTKkRpZ2lDZXJ0IEdsb2JhbCBH
MyBUTFMgRUNDIFNIQTM4NCAyMDIwIENBMTB2MBAGByqGSM49AgEGBSuBBAAiA2IA
BHipnHWuiF1jpK1dhtgQSdavklljQyOF9EhlMM1KNJWmDj7ZfAjXVwUoSJ4Lq+vC
05ae7UXSi4rOAUsXQ+Fzz21zSDTcAEYJtVZUyV96xxMHOGwYF2zK28cLJlYujQf1
Z6OCAYIwggF+MBIGA1UdEwEB/wQIMAYBAf8CAQAwHQYDVR0OBBYEFIoj655r1/k3
XfltITl2mqFn3hCoMB8GA1UdIwQYMBaAFLPbSKT5ocXYrjZBzBFjaWIpvEvGMA4G
A1UdDwEB/wQEAwIBhjAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwdgYI
KwYBBQUHAQEEajBoMCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC5kaWdpY2VydC5j
b20wQAYIKwYBBQUHMAKGNGh0dHA6Ly9jYWNlcnRzLmRpZ2ljZXJ0LmNvbS9EaWdp
Q2VydEdsb2JhbFJvb3RHMy5jcnQwQgYDVR0fBDswOTA3oDWgM4YxaHR0cDovL2Ny
bDMuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0R2xvYmFsUm9vdEczLmNybDA9BgNVHSAE
NjA0MAsGCWCGSAGG/WwCATAHBgVngQwBATAIBgZngQwBAgEwCAYGZ4EMAQICMAgG
BmeBDAECAzAKBggqhkjOPQQDAwNoADBlAjB+Jlhu7ojsDN0VQe56uJmZcNFiZU+g
IJ5HsVvBsmcxHcxyeq8ickBCbmWE/odLDxkCMQDmv9auNIdbP2fHHahv1RJ4teaH
MUSpXca4eMzP79QyWBH/OoUGPB2Eb9P1+dozHKQ=
-----END CERTIFICATE-----
---
Server certificate
subject=C=US, ST=California, L=Los Angeles, O=Internet Corporation for Assigned Names and Numbers, CN
issuer=C=US, O=DigiCert Inc, CN=DigiCert Global G3 TLS ECC SHA384 2020 CA1
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 2725 bytes and written 406 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Protocol: TLSv1.3
Server public key is 256 bit
This TLS version forbids renegotiation.
Compression: NONE
Expansion: NONE
```

```
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID: 8A44040B3580E25E7DDF7B6BAE44B6CD5FDEB20DEEAE5770F1C490D6732935B3
    Session-ID-ctx:
    Resumption PSK: 20D378F8CC589EFF0D368D4E594EA7408BDC38C8FEFFAF626A99A5FF55B1A9A03A3C5FE48EC47ED84
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 83100 (seconds)
    TLS session ticket:
    0000 - 00 04 20 7f cf e9 26 ee-e5 69 98 bc c0 58 68 f3   .. ...&..i...Xh.
    0010 - a4 1b 23 b0 04 26 82 fe-34 82 7a 36 4c 0e 6b 3f   ..#..&..4.z6L.k?
    0020 - 94 4e 47 db 2d 62 d5 23-54 5e ce 5f c8 8a 05 20   .NG.-b.#T^._...
    0030 - af f3 f9 c5 85 aa 3c 43-e6 95 d0 02 bc 16 e0 6c   ......<C.......l
    0040 - 22 af 33 34 a0 ec 4f 3d-48 b7 7e f0 1b 64 f3 e5   ".34..O=H.~..d..
    0050 - 01 fa 89 77 03 f9 00 e4-4c 9d d8 d1 8f 7c eb 39   ...w....L....|.9
    0060 - c4 dc ef 67 15 59 e4 3e-85 ad 66 bd 74 13 b0 5b   ...g.Y.>..f.t..[
    0070 - e2 5e b0 b2 00 a8 cc 1c-80 76 1f 96 46 41 92 5b   .^.......v..FA.[
    0080 - 57 30 5f 82 39 87 3a 00-8b 87 0c 90 dd 15 59 f9   W0_.9.:.......Y.
    0090 - 71 d0 43 18 9f a8 6a 32-8a da 58 b8 fc b8 75 d3   q.C...j2..X...u.
    00a0 - 73 10 7a fc e8 53 80 21-0a c3 20 56 29 08 32 e9   s.z..S.!.. V).2.
    00b0 - 73 2b f1 fb 9b 8d 4f b8-44 d3 a0 1f 69 3d 45 9f   s+....O.D...i=E.
    00c0 - 47 f2 83 c9 3c b6 ff f8-5d 5c 8a c7 39 fd 34 3d   G...<...]\..9.4=
    00d0 - 55 56 83 7a 23 3d b0 f6-04 9f c1 e5 b3 03 cf b2   UV.z#=..........

    Start Time: 1740900857
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: no
    Max Early Data: 0
---
read R BLOCK
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID: 3ADC301F87AD3C32707C5578A8FF66AA0EC41DCC27B1455E4202B0834E7B5941
    Session-ID-ctx:
    Resumption PSK: D2201114D1FCD86FB4B6F60FC0E8B64FF006CC5396261676F35845C6B651227DB3EC3B3E211FCCF24
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 83100 (seconds)
    TLS session ticket:
    0000 - 00 04 20 7f cf e9 26 ee-e5 69 98 bc c0 58 68 f3   .. ...&..i...Xh.
    0010 - 4b a3 7e 60 08 24 3d 34-b1 2e f3 e2 ae 2e c7 83   K.~`.$=4........
    0020 - ff 41 b3 7b 35 e0 4b 24-45 74 9a 7c 1c 36 91 64   .A.{5.K$Et.|.6.d
    0030 - 37 f9 a1 ac 26 46 2c 72-58 2f 9b de 35 3a 6b d7   7...&F,rX/..5:k.
    0040 - 41 81 c7 ad 3b a5 19 11-d8 5d 9b 50 fb 83 73 de   A...;....].P..s.
    0050 - 87 cf 23 6e f1 cb 3e fb-da 8c 8c 66 40 06 88 44   ..#n..>....f@..D
    0060 - 74 47 1b f5 67 4b 82 63-22 99 c7 a0 1d af 86 e9   tG..gK.c".......
    0070 - 89 f9 47 37 83 24 ae 2c-82 41 a5 3f d6 90 fa 4b   ..G7.$.,.A.?...K
    0080 - b7 33 86 ff 43 b2 d1 db-9d d0 6a 35 87 7a 0e 12   .3..C.....j5.z..
    0090 - 05 d2 13 5d 62 87 e5 02-39 ff 4f f6 e9 77 72 77   ...]b...9.O..wrw
    00a0 - 62 5a 3b bb 06 b8 d2 d0-67 e8 9d 67 f1 b2 b5 ee   bZ;.....g..g....
    00b0 - 8f 4d 66 e9 13 20 a7 79-2e 7e 1a 97 d1 87 27 6d   .Mf.. .y.~....'m
    00c0 - 82 9a 49 8f 16 48 f3 1d-5c b8 83 6f 70 29 98 e1   ..I..H..\..op)..
    00d0 - 40 c7 15 3a 28 56 36 af-15 81 40 e6 49 3b 5c c7   @..:(V6...@.I;\.

    Start Time: 1740900857
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: no
    Max Early Data: 0
---
read R BLOCK
closed
```

2. Save BEGIN CERTIFICATE to END CERTIFICATE into separate files called `ca_cert.pem` and `server_cert.pem`.

Task 6 - Issues

The lab instruction assumes that the web server uses RSA for the signature. However, at the time writing this, it uses ECDSA. Hence, the steps in the lab instructions are not reproducible anymore.

As proof:

```
$ openssl x509 -in ca_cert.pem -text -noout | grep Signature
    Signature Algorithm: ecdsa-with-SHA384
            Digital Signature, Key Agreement
                Signature : ecdsa-with-SHA256
                Signature : ecdsa-with-SHA256
                Signature : ecdsa-with-SHA256
Signature Algorithm: ecdsa-with-SHA384
Signature Value:
```