

# Lab 1: Public Key Infrastructure

Shayona Basu

CMSC 23206

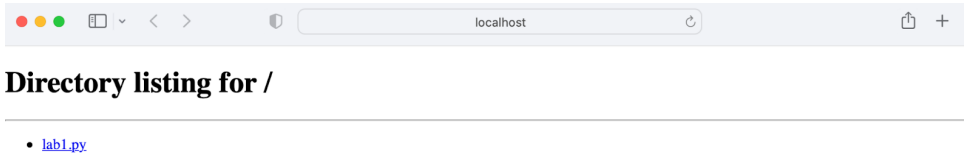
Prof. Feamster

## 1. Host a local web server

At first I just used the `http.server` library in Python, but I wasn't able to find any packets being sent to it in Wireshark. I added filters such as `ip.addr == 127.0.1.0` to search specifically for my local host server, but Wireshark was not capturing it and I wasn't sure why. To be specific, I created the server, by simply putting the following in terminal,

```
$ python -m http.server
```

And this is how the server looked,



From what it looks like, it just displayed the folder in the current directory. This was a very simple approach, which may be why it didn't work?

I had moved on to step 2 and step 3, and actually managed to assign a certificate to this local host and get it signed, but as I wasn't able to capture any packets or also properly check that the http changed to https, I decided to try again using Flask.

The following is the code I used to host my server using Flask:

```
from flask import Flask, render_template
app = Flask(__name__)

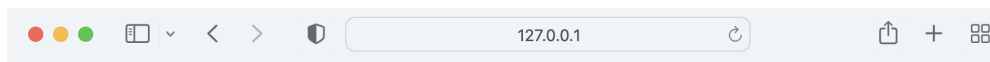
@app.route("/")
def home():
    return render_template("test.html")
app.run(debug=True)
app.run(host="127.0.0.9", port=8080, debug=True)
```

*Note: `test.html` is just some standard html code used for testing. I added some unique words to try and make it easier to spot on Wireshark when going through the packets. Due to Flask specifications, I had to put `test.html` in a `templates` folder in my directory.*

I ran this file on terminal, and got the following output:

```
shayonbasu@Shayonas-MacBook-Pro lab1 % python test.py
* Serving Flask app "test" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with watchdog (fsevents)
* Debugger is active!
* Debugger PIN: 863-179-755
127.0.0.1 - - [21/Oct/2023 10:42:19] "GET / HTTP/1.1" 200 -
```

We look at the line where it says 'Running on' and we see it is running on our local server using port 5000. To view our server, we type in '<http://127.0.0.1:5000/>' into our browser (I used Safari).



## azerbaijan

some classified information that we will try and find because it is not encrypted in http, but is in https  
slay

We can see that the server is running and is showing the text.

## 2. Identify why HTTP is not secure

- Explain how an eavesdropper can “sniff” web traffic between a client and HTTP server to see what is being communicated (ie. which resources are being fetched and the contents of the resources)
- Use Wireshark to capture network traffic between your local web server and a local client**

### Question 2a)

HTTP is a protocol web servers use to communicate between web servers and clients. However, HTTP is not inherently secure, and it lacks the mechanisms required to protect data from unauthorized access and eavesdropping. HTTP operates on an open, text-based format, which makes it susceptible to eavesdropping. The main reason why HTTP isn't inherently secure is because there is a lack of encryption and no authentication is required.

Further, data sent using HTTP is not encrypted, making it easy for eavesdroppers to intercept and read. Without encryption, sensitive information, such as login credentials and personal data, is at risk. Further, HTTP does not have a mechanism for verifying the authenticity of the web server. As a result, malicious actors can impersonate a legitimate server and trick clients into sending data to them.

As we discussed in class, HTTP servers can be especially vulnerable to Man-in-the-Middle Attacks: Eavesdroppers can employ man-in-the-middle (MitM) attacks to intercept and modify data exchanged between the client and server. By positioning themselves between the two parties, they can capture and tamper with the communication without the client or server's knowledge. Eavesdropping on web traffic between a client and an HTTP server is a real concern, especially when the data traverses insecure or untrusted networks. Your notes touch upon the potential risk associated with DSL (Digital Subscriber Line) connections, which use existing telephone lines for high-speed internet access.

A little bit more about the Physical vulnerabilities of DSL is that DSL connections share the same copper infrastructure as traditional telephone lines. This infrastructure can be accessed and compromised by individuals with physical access to the wiring. Eavesdropping on DSL traffic typically requires specialized equipment and knowledge. An eavesdropper would need to access the DSL Access Multiplexer (DSLAM) or similar hardware to intercept the data.

To "sniff" web traffic, an eavesdropper may use tools like Wireshark to capture packets passing through the network interfaces, enabling them to analyze the data. The captured traffic could include various protocols such as SSH, FTP, and HTTP. For HTTP, this means the eavesdropper can see the URLs being accessed, the content of web pages, and any data exchanged between the client and server. This lack of encryption in HTTP further exacerbates the risks associated with eavesdropping.

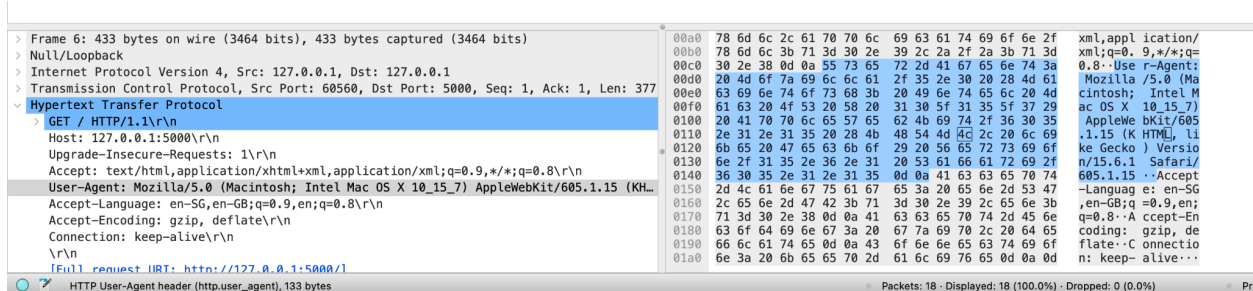
### Question 2b)

After installing Wireshark I played around to try and understand that each row were packets being delivered to different addresses, and how powerful filtering could be.

I tried to go into each network to test the packets in my first runthrough, but then I researched to try and understand what each network actually captured. I then understood **Loopback: lo** was the most appropriate as my server is running on my local machine, and it can capture packets sent through which my web server and client communicate.

I started capturing on Wireshark, then I went on Terminal and ran my server again. I went to Safari and went to where it was being hosted, and clicked around a bit. I wish I had a user input feature, because I think that it would be easier to capture what the user was typing in when I would search on Wireshark later. Anyway, I started just clicking through the packets and then found my html request being sent through a get request! I saved and exported it to my lab1 folder that I am pushing to Git, but here is also a picture:

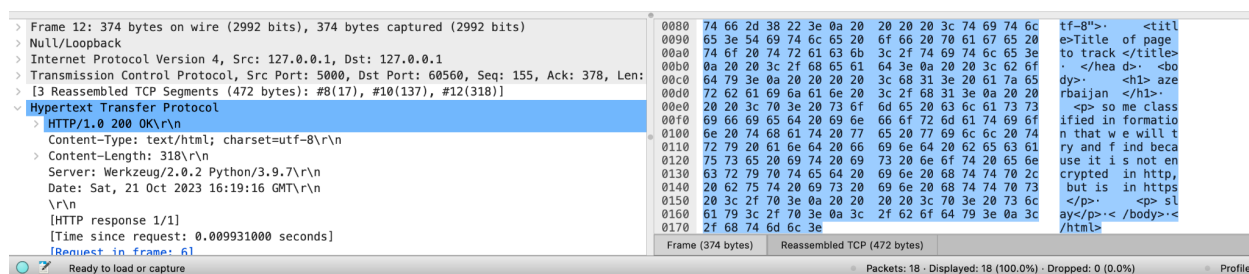
The following packet is I think the packet hearing me requesting to start the server:



The image shows a Wireshark packet capture window. The left pane shows the packet list with 'Frame 6: 433 bytes on wire (3464 bits), 433 bytes captured (3464 bits)' selected. The middle pane shows the packet details for 'Hypertext Transfer Protocol' and 'GET / HTTP/1.1'. The right pane shows the raw packet data in hexadecimal and ASCII. The ASCII column shows the following text: 'xml,application/xml;q=0.9,\*/\*;q=0.8; User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.6.1 Safari/605.1.15'. The packet is an HTTP GET request for the root path.

```
> Frame 6: 433 bytes on wire (3464 bits), 433 bytes captured (3464 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 60560, Dst Port: 5000, Seq: 1, Ack: 1, Len: 377
> Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Host: 127.0.0.1:5000\r\n
    Upgrade-Insecure-Requests: 1\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/15.6.1 Safari/605.1.15
    Accept-Language: en-SG,en-GB;q=0.9,en;q=0.8\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
    \r\n
    [Full request URI: http://127.0.0.1:5000/]
  HTTP User-Agent header (http.user_agent), 133 bytes
```

This next one, you can see the HTML text in the window:



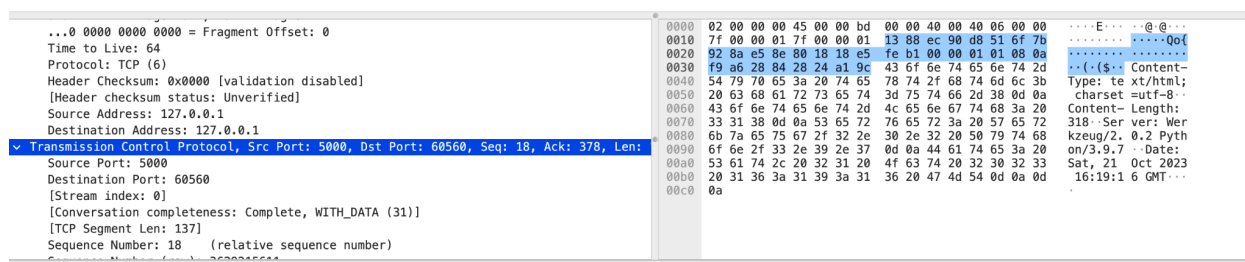
highlighted text on the right you can see “Title of page to track </title> ... some classified information that we will try and find because it is not encrypted in https but is in http”

The two above packets were Protocol: HTTP, but I also found TCP protocol packets, shown below:

This is the overview. You can see it is being sent from 60560 → 5000 ports. Where 5000 port is where my server is being hosted.

9	15.650552	127.0.0.1	127.0.0.1	TCP	56	60560 → 5000	[ACK] Seq=378 Ack=18 Win=408256 Len=0 TSval=673489308 TSecr=4188416130
10	15.652425	127.0.0.1	127.0.0.1	TCP	193	5000 → 60560	[PSH, ACK] Seq=18 Ack=378 Win=407872 Len=137 TSval=4188416132 TSecr=673489308 [TC
11	15.652446	127.0.0.1	127.0.0.1	TCP	56	60560 → 5000	[ACK] Seq=378 Ack=155 Win=408128 Len=0 TSval=673489310 TSecr=4188416132

Here is the enlarged view:



### 3. Create a self-signed certificate and upgrade your web server to HTTPS

- Generate an SSL certificate for your web server, add the certificate to your list of locally trusted roots, and restart the web server with the certificate
- Include a network track (captured using Wireshark) and comment on the differences between the contents of HTTP and HTTPS (TLS) traffic

#### Question 3a)

I basically followed this tutorial exactly:

<https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/>

This tutorial allowed me to bypass the “Your connection is not private” error in Chrome by accessing a local site via HTTPS and the certificate is not configured. It allowed me to bypass this by showing how to become a ‘Tiny’ Certificate Authority, and accept the certificate in local devices using Keychain on Mac.

The main steps are the following:

#### 1. Generating the Private Key and Root Certificate on macOS

- Basically used openssl, which I had to download
- I created a /certs folder to store the local certificate files

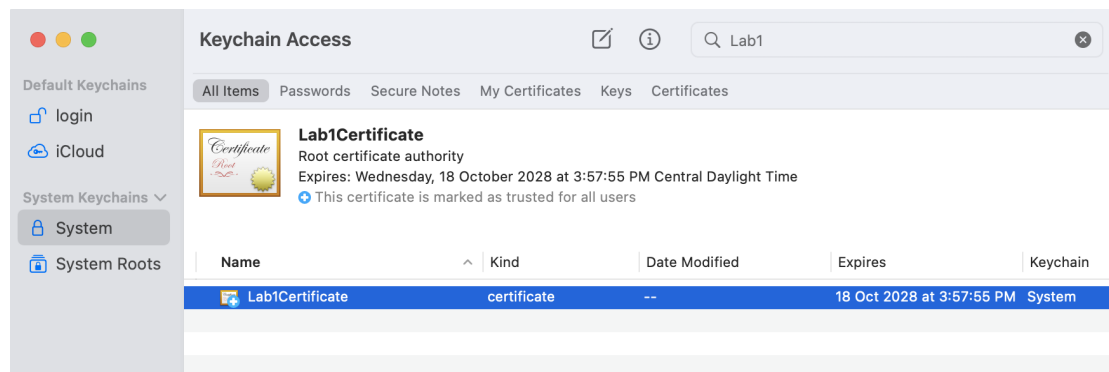
- c. **Generating private key by** `openssl genrsa -des3 -out myCA.key 2048`
  - i. It asked for a password and I put 'feamster'
- d. **Generate a root certificate** which prompted the password I just chose and a list of questions to create the certificate. I am pasting the terminal output below

```
(base) shayonabasu@Shayonas-MacBook-Pro certs % openssl req -x509 -new -nodes -key
myCA.key -sha256 -days 1825 -out myCA.pem
Enter pass phrase for myCA.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Illinois
Locality Name (eg, city) []:Chicago
Organization Name (eg, company) [Internet Widgits Pty Ltd]:FeamstersClass
Organizational Unit Name (eg, section) []:UChicago
Common Name (e.g. server FQDN or YOUR name) []:Lab1Certificate
Email Address []:shayona@uchicago.edu
(base) shayonabasu@Shayonas-MacBook-Pro certs % openssl genrsa -out feamster.test.key
2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
```

What to look at is that I created the certificate 'myCA.pem', and the Common Name (or name of the certificate is 'Lab1Certificate'. *Note: SHA256 encryption*

## 2. Adding the Root Certificate to Mac Keychain

- a. I used the macOS Keychain App to import myCA.pem and searched for my Certificate 'Lab1Certificate'



And made sure to mark it as trusted for all users.

## 3. Creating CA-Signed Certificates for Your Dev Sites

- a. Now the certificate is authorized on all our devices, and we can **sign the certificate for any new sites we are hosting that need HTTPS**

- b. To do this, we need to create a private key for the dev site, through a similar process in step 1. Here is the terminal input and output:

```
(base) shayonbasu@Shayonas-MacBook-Pro certs % openssl genrsa -out feamster.test.key
2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
(base) shayonbasu@Shayonas-MacBook-Pro certs % openssl req -new -key feamster.test.key
-out feamster.test.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IL
State or Province Name (full name) [Some-State]:IL
Locality Name (eg, city) []:Chicago
Organization Name (eg, company) [Internet Widgits Pty Ltd]:UChicago
Organizational Unit Name (eg, section) []:FeamsterLab
Common Name (e.g. server FQDN or YOUR name) []:Feamster
Email Address []:shayona@uchicago.edu

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:feamsterkey
An optional company name []:.
```

The tutorial mentioned that we are running openssl x509 because the x509 command allows us to edit certificate trust settings. In this case we're using it to sign the certificate in conjunction with the config file. We can configure local web servers to use HTTPS with the private key and the signed certificate.

After running the command to create the command, we have:

1. **The private key** feamster.test.key
2. **The certificate signing request (csr)** feamster.test.csr
3. **Signed certificate** feamster.test.crt

## 4. Configuring Local Web Server to use HTTPS with the Private key and Signed Certificate

I used the following bash script from the tutorial (I placed the bash script in the ~/certs folder)

```
#!/bin/sh

if [ "$#" -ne 1 ]
then
echo "Usage: Must supply a domain"
exit 1
fi

DOMAIN=$1

cd /Users/shayonabasu/Security/security-course/lab1/certs

openssl genrsa -out $DOMAIN.key 2048
openssl req -new -key $DOMAIN.key -out $DOMAIN.csr

cat > $DOMAIN.ext << EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment,
dataEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = $DOMAIN
EOF

openssl x509 -req -in $DOMAIN.csr -CA ./myCA.pem -CAkey ./myCA.key
-CACreateserial \
-out $DOMAIN.crt -days 825 -sha256 -extfile $DOMAIN.ext
```

Lastly,

Terminal:

```
(base) shayonabasu@Shayonas-MacBook-Pro ~ %
/Users/shayonabasu/Security/security-course/lab1/certs/babash.sh
"/Users/shayonabasu/Security/security-course/lab1/test.py"
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
```

```

.....+++++
e is 65537 (0x010001)
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:feamsterkey
An optional company name []:
Signature ok
subject=C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
Getting CA Private Key
Enter pass phrase for ./myCA.key:

```

Worked! Now, I have to update my test.py to the following:

```

'''
test.py
'''

from flask import Flask, render_template
from OpenSSL import SSL

app = Flask(__name__)

# Create an SSL context and load your SSL certificate and private key

def get_passphrase(*args):
    return "your_passphrase"

context = SSL.Context(SSL.SSLv23_METHOD)
context.set_passwd_cb(get_passphrase)
context.use_privatekey_file('test.py.key')
context.use_certificate_file('test.py.crt')

@app.route("/")
def home():

```



```

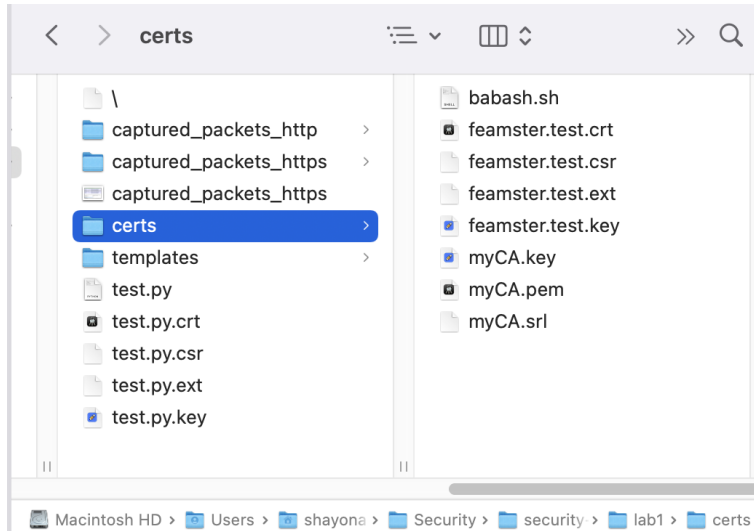
return render_template("test.html")

app.run(host="127.0.0.1", port=8443, debug=True, ssl_context = context)

```

Just note adding the OpenSSL library, and connecting to the just now created test.py.key and test.py.crt files. Also, note how we have to include the get\_passphrase argument because our certificate has a passphrase. Lastly, note how we added ssl\_context in the app.run file, as well as choosing 8443 port, which is an unprivileged port.

Here is how my folder looks, you can see the 'test.py.crt' is created:



Here is the output after running test.py in terminal:

```

python test.py
* Serving Flask app "test" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on https://127.0.0.1:8443/ (Press CTRL+C to quit)
* Restarting with watchdog (fsevents)
* Debugger is active!
* Debugger PIN: 863-179-755
Exception in thread Thread-2:
Traceback (most recent call last):
  File "/Users/shayonabasu/opt/anaconda3/lib/python3.9/threading.py", line 973, in
_bootstrap_inner
    self.run()
  File "/Users/shayonabasu/opt/anaconda3/lib/python3.9/threading.py", line 910, in run
    self._target(*self._args, **self._kwargs)
  File "/Users/shayonabasu/opt/anaconda3/lib/python3.9/site-packages/werkzeug/serving.py", line
950, in inner
    srv = make_server(
  File "/Users/shayonabasu/opt/anaconda3/lib/python3.9/site-packages/werkzeug/serving.py", line
782, in make_server
    return ThreadedWSGIServer(

```

```
File "/Users/shayonbasu/opt/anaconda3/lib/python3.9/site-packages/werkzeug/serving.py", line 708, in __init__
    self.socket = ssl_context.wrap_socket(self.socket, server_side=True)
AttributeError: 'Context' object has no attribute 'wrap_socket'
```

And, here are the Wireshark Packets

Running on my local machine!!!

I think it's now encrypted?

Total Length: 64  
Identification: 0x0000 (0)  
010. ... = Flags: 0x2, Don't fragment  
... 0000 0000 0000 = Fragment Offset: 0  
Time to Live: 64  
Protocol: TCP (6)  
Header Checksum: 0x0000 [validation disabled]  
[Header checksum status: Unverified]  
Source Address: 127.0.0.1  
Destination Address: 127.0.0.1  
Transmission Control Protocol, Src Port: 55613, Dst Port: 8443, Seq: 0, Len: 0  
Source Port: 55613  
Destination Port: 8443  
(Stream index: 0)

Dst Port is what we specified, so we know its outs: 8443

0000 02 00 00 00 45 00 00 00 00 00 00 00 00 06 00 00 ... 0 0 0  
0010 7f 00 00 01 7f 00 00 01 d9 3d 20 fb 36 c3 1f 3f ... 6  
0020 00 00 00 00 b0 02 ff ff fe 34 00 00 02 04 3f d8 ... 4  
0030 01 03 03 05 01 01 08 0a fe b6 3e d4 00 00 00 00 ...  
0040 04 02 00 00