FCS Assignment-2

-Ishmeet Kaur(2015042)

Note: All screenshots and files are arranged in their respective folders to keep the report clean.

Part-1

1. Yes, IIIT-Delhi does follow an authentication system.
First, the **client**(student, faculty or any other user) have to login through a firewall(The **authenticator**). Then, the **authentication server** will verify the credentials and only then can a user enter the IIIT-D network.
2. Yes, packet Sniffing can be done through various software like **Wireshark** and burp suite. which can even change the packet data. Such packet-analyser tools let the users see what's happening on the network at a microscopic level, and decrypt those packets and modify them as well.
3. A typical session can simply be going to gmail, backpack, social networking sites, erp etc. A typical internet usage session refers to logging into multiple social network sites like facebook and twitter, multiple networks like email, erp etc. Sniffing here can lead to **sensitive information like passwords being leaked, data can be modified**, which would be very risky from the college point of view in case of erp, as it contains the grades of students.

Part-2

1. Injecting the javascript in the browser, **an alert appeared saying hello world**. The javascript we entered is interpreted as the website that the browser has to display for us. This is very risky, as this means, we can change the behavior of the website according to us, just by running simple javascript functions.

2.
javascript:void(document.getElementById('logo').src='https://tctechcrunch2011.files.wordpress.com/2017/08/horror-movie.png?w=1279&h=727&crop=1');
This **changed the image displayed as the logo**. **The source code remains the same**, just that from the user point of view, he is seeing a different image. This shows the power of javascript, how it can change the dom elements. **Simply access the element you want to modify, and on it write your own javascript function.** An attacker can do some bank transfers and exploit the power of javascript this way with just one line of code.

3. After clicking the check out this cool link the **logo gets changed** to the image that is given as new source in the javascript function. But **since, the image is the same, no changes are reflected**.
After changing the source to :
https://www.wpblog.com/wp-content/uploads/2017/08/wordpress-site-is-hacked.jpg
Now this new image is displayed as the logo. **Whenever we click the link, logo image changes.**

4.
<a href="javascript:void(document.getElementById('logo').src='https://w ww.wpblog.com/wp-content/uploads/2017/08/wordpress-site-is-hacked.jpg');" >
On adding this, automatically, the logo gets changed on clicking the link.

5. On clicking the link for go to IIITD homepage, **we get to iiitd.ac.in as expected**. Then, when first click on 'Hello, see this page' link, nothing happens. But then on clicking 'Go to IIITD' again, instead **we reach google.com**
Source code is the same
The 'a' element has been tampered with on clicking the 'Hello see this page link', which booted a javascript function to instead go to google.com
Similarly, a hacker can trick the users to visit malicious sites by tampering the links, perform evil functions on clicking links which the user thinks are non suspicious.

6.
** Check your email**
Hackers can trick the users by creating dummy websites which appear exactly the same, where the user would enter his username and password, which the attacker would steal. This is called phishing.

7. Yes the session ID is set correctly.
<a
href="javascript:window.location='http://defencely.com/blog/wp-content/uploads/2013 /
06/ways-hackers-hack-your-website-e1371080108770 >
On clicking the link, the attacker takes us to a malicious site with users session Id attached to the link, thus stealing our data using XSS vulnerability.

8. There are a no. of techniques **to prevent XSS attacks**:
=> Escape the HTML before inserting the untrusted data into HTML Element Content
=> Never Put untrusted data in locations that are not allowed
=> Always use attribute escape before adding data in HTML attributes
=> JavaScript escape before inserting untrusted data into javascript data values
=> Prevent DOM-based XSS by keeping variable checks.

=> CSS escape and strictly validate before inserting untrusted data into HTML style property values
=> Clean up the HTML markup with a necessary library

In our case, for eg, we can Escape HTML by a function which returns:
htmlWeEnter.replace(/&/g, '&').replace(/</g, '<').replace(/"/g,"');
9. Command used to create a self-signed certificate:
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout ishi.pem -out ishi.pem
Now we can host it using any server, like Go.

10. **No, HTTPS cannot prevent XSS attacks**. These attacks are different, they are based on the frontend vulnerabilities.

Part-3

1. For this part, I have used **Burp Suite**
First, we setup a proxy at 127.0.0.1:8080, (where burp suite is running).
1.1. In chrome go to Settings. Search Proxy, click on open Proxy Settings, setup a http proxy with address as 127.0.0.1 and potn no. as 8080
1.2. Open burp suite, run in default configuration.
1.3. The library site of IIITD is not secure. Go to library.iiitd.edu.in.
1.4. We see burpsuite has already started capturing our packets.
1.5. Try to log in to by entering the username and password.
1.6. **The username and password are now visible in plain text in burp suite**
1.7. This is highly insecure, also **the files we download also go through the intereceptor** in the burp suite.

2. The vulnerabilities are:
2.1 **Data Leakage**- All the passwords and usernames can be easily accessed by the hackers.
2.2 **Denial of Service(DOS)**-The interceptor can keep the packets and not send it forward thus denying the access to the information.
2.3 **Tampering of data**-The interceptor can edit the downloaded files and then forward the packet,thus passing wrong information.

3. One should always use **https protocol** which involves a secure ssl handshake between the server and client.To prevent the misuse of data, user should not give permissions to any such interceptor. Also **avoid usage of any kind of proxy in your system**. **IpSec should also be used at the network layer, and public/private key cryptosystem for key exchange is a must.**

Part-4

1. a)
=> Get the system current date-time.
=> Encode it in utf-8
=> Hash this value using hashlib's sha256. Store this hash value
=> Take the integer value of this hexadecimal digest and then chop-off the last four digits of the hash using modulo operations . This is my function F. **F=Hash%10000**
=> This is the necesary otp.
=> Now take user input as 4 digit otp
=> Match the input with the output of F on hash
=> If they match, OTP is verified, else wrong OTP is entered
I chose this function F as it is ensures the least no. of collisions. SHA256 will generate unique(almost) hashes for us and taking the last 4 digits makes the most sense than any other logic to keep the OTPs as unique as possible.

b) No approach is 100% secure. But, this is very close to it.
Like all OTP-based schemes, this approach is still vulnerable to **session hijacking**, i.e. attacking the session after user has logged in.
Since we are using **only 4 digits** of OTP, it might happen in the very rare case, that OTPs are same.
If two people **request for OTP at the same exact time**, then there isn't much we can do in this approach.
OTP codes can be phished just as passwords can, though they require attackers to proxy the credentials in real time rather than collect them later on in time. That's why this method is considered safer than directly logging in via username and password.
Many a times, people use XOR and other approaches to generate F. But, they have a much higher chance of collision as more or less, two numbers will be different in the hash, hence XOR would mostly be 1. Some people use sum and averages, but they also have a higher chance of collision. Overall, **this method ensures approximately no collisions.**

2. **First create a certificate** using the command openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout ishi2.pem -out ishi2.pem
Run the server code: gcc-Wall-o server server.c-L/usr/lib-lssl -lcrypto
Then type - sudo ./server 8080 where 8080 is the port number.
Run the client code : gcc -Wall -o client client.c -L/usr/lib -lssl-lcrypto
Then type - ./client 127.0.0.1 8080
This chat app return a valid message or error message after taking input from the client. With the help of WireShark we can verify if the data sent is being encrypted. Screenshots and files attached.

3.
=> Download Tor using **brew install tor** (Mac OS equivalent of apt-get install)
=> This will provide us the necessary torcc(tor configuration file) file.
=> Run command tor
=> Now **tor is running at 127.0.0.1 and port 9050**
=> Set the **socks proxy** in proxy settings
Tor is now running!

=> **Uncomment these lines,in the torcc files:**
#SOCKSPort 9050 # Default: Bind to localhost:9050 for local connections.
#SOCKSPort 192.168.0.1:9100

=>**To change the permanent connections for tor, uncomment**
#SOCKSPolicy accept 192.168.57.82
#SOCKSPolicy reject *

=> **To use bridge, uncomment**
#BridgeRelay 1
#UseBridges 1

4.
The command used to run a cipher file is:
time openssl <algo_name> -in largefile.txt -out <output_file>
Here is the output for each algorithm:

1. RC4
real 0m5.018s
user 0m0.070s
sys 0m0.038s

2. AES256
real 0m6.005s
user 0m0.092s
sys 0m0.045s

3. RSA: error

RC4 and AES both are quite fast. But, RSA gives us error as it cannot encrypt such a big file.To solve the issue we can chunk the file in accordance with the key size or we can create a large enough key to encrypt the file in one go. Stil, RSA would be slow.
Stream ciphers encrypt the file bit by bit wheras block ciphers encrypt the data per block which makes stream ciphers relatively faster.