TLS Assignment

Samir Farhat Dominguez - U41707119

April 4, 2022

1 Review Questions

1.1 Name three ways for an attacker to intercept network traffic that we have learned about in class (each way should be exploiting a different protocol.)

1. ARP Cache Poisoning attacks exploits the lack of authentication in the UDP protocol.
2. DNS Spoofing exploits UDP, in addition to a lack of authentication in DNS Authority and Reply protocols.
3. The Downgrade attacks targets several weaknesses in TLS1.2 handshakes, and any TLS system that supports outdated OpenSSL security protocols.[3]

1.2 How does TLS reduce the harms an adversary can induce by intercepting network traffic?

TLS 1.3 includes several considerations and nuances to mitigate damage in case of interception of network traffic. Speaking generally, first and foremost, TLS1.3 has been trimmed to support cipher-suites that meet the criteria of not having vulnerabilities exposed(at least at the time of introduction) and have perfect forward secrecy. That means that all traffic intercepted after the handshake sequence is encrypted in such a way that it is either impossible unfeasible to decrypt under formal definitions such as CPA.[1] Perfect forward secrecy is beneficial, as the constant change of keys ensures that even if keys are compromised, the quantity of data decipherable from this compromise is minimized. Moreover, TLS1.3 gets rid of sign-then-encrypt in favor of encrypt-then-sign, as the former permitted adversaries to gain information from rejected messages, and pose as a legitimate party by spoofing a signature.[2]

The principal ways that network traffic interception damage is mitigated relates to the following. First, are the nonce messages that both the client and server need to to produce to build keys and parameters. This makes it such that both incoming and outgoing traffic needs to be intercepted to be able to decipher keys and secrets, essentially doubling the difficulty of the problem. Moreover, the `ClientFinished` and `ServerFinished` messages. In the Handshake part of these messages is the hashing of the entirety of previous communications, meaning that building these for both the Server and Client to not notice alterations is tremendously complicated.[2]

1.3 Consider a flawed TLS implementation where the client "forgets" to check the signature on the server's certificate

The principal goal achieved by signatures and MACs is authenticity. Essentially, performing a signature on a sequence of bytes generates some message of some length designated as σ. This σ should be irreproducible by a party that does not have access to the secret key and the native PRF producing the signature.[4] So, in the TLS scheme if the client does not check the signature then authenticity is lost, and the client will accept all messages sent to it, thereby permitting the following attack.

In the ephemeral Diffie-Hellman key exchange when client certificates are not required(and thus client signatures to authenticate that certificate) an adversary that is able to intercept messages can do the following. First the client sends the `ClientHello`, which is read by adversary and sent to the server, the server sends the `ServerHello`, which again is intercepted to be read and then forwarded to Client.[1] Then, the adversary can send a self-signed certificate to the client; since signatures are not checked, no flags are raised, TLS proceeds and the actual server certificate is intercepted and discarded by the adversary. Assuming an ephemeral Diffie-Hellman, the adversary intercepts the server's g^b reading it, produces some arbitrary g^c, and sends it to the server. After, the adversary sends its own `ServerHelloDone` to the client, who responds with its own g^a, and the adversary reads, discards, and sends its g^c. The client believes this is the server's g^b and the server believes this is the client's g^a. As such, the adversary is able to compute g^{ac} for client-adversary communications and g^{bc} for adversary-server communications. From there, the adversary is able to communicate ad-infinitum intercepting messages, modifying them and using its sets of keys to send whatever it likes with the server and client. At the end of TLS communications, it is able to recreate all the messages it sent to either party, so the `Finished` message is verified for the client and server as all the messages either party thinks the other sent are in the `opaque verify_data[verify_data_length];` structure.[2]

2 Tinkering with TLS

2.1 Imagine a version of TLS without rc.

The client-generated random structure is a set of 28 bytes generated from a secure random number generator and 4 bytes designating the unix timestamp. This random value is part of what is used to compute the master secret as well as Diffie-Helmann parameters in case of ephemeral DH. In TLS, the server is designed to abort and terminate client-hellos and key parameter messages that are repeated.[1] However, so the server doesn't have to be responsible for holding all past messages, the `ClientHello` messages are recorded and have a mapped `obfuscated_ticket_age`, related to their unix timestamp[1]. Essentially the server ties a message to a expected period of time, discarding it once the expected period of time has elapsed. If a `client.random` is not checked, the server has no viable way for the server to determine a replay attack has been initiated. So an attacker may be able to get through the initial part of a replay attack.[1] However, where the rubber meets the road, and a replay attack would be greatly weakened is in the `server.random`. Unless the server's PRF is not up to scratch, the server will designate entirely unique parameters to create the keys and master secrets, as it is using its own `server.random` to do so. This means previously used keys would not be resent.[6] As such deciphering previous communications would be unfeasible and the damage a replay attack could do is greatly mitigated. However, a replay attack to resume a session is possible, as keys are reused when sessions are resumed to reduce handshake times, but due to resumptions not involving key exchanges, all the attacker would be able to do is repeat previous messages hoping something productive happens.[7] For instance if a credit card was charged in a message, then they can charge it once again but won't really understand what is going on.[8]

1. Proper TLS handshake takes place between client and server, where the attacker consumes all messages sent for information gathering.
2. Attacker sends `ClientHello` with same session id, assuming it will still be in the Server cache, and will be unable to reject due to lack of repeated r_c
3. Client consumes `ServerHello` with same session id
4. As per the protocol, `ChangeCipherSpec` is sent and application data starts flowing
5. Adversary repeats all previously seen messages, hoping to gather information or do something prodcutive to its goals
6. He can try repeat and reorder arbitrary number of times, but again they are just hoping and praying.

2.2 Imagine a version of TLS without rs

The omission of a `ServerHello.random` is much more problematic, as an attack on the ephemeral Diffie-Hellman process becomes possible. The RFC explicitly states how it utilizes the nonce to hash the temproary Diffie-Hellman parameters before signing to avoid replays.[1] As such, if an adversary performs a replay attack with the same client-random nonce, even with an altered unix timestamp then the the keys, parameters, and master secret will match

up with previous communications.[1] This will expose previous keys to the adversary, thereby exposing previous communications and allowing future illegitimate communications with the server. This plays into resuming previous sessions as well. If the client uses the same `client.random` and the same session ID than for a previous handshake, the server is only able to assume a previous session is being resumed, and continue exchanging application layer data that the adversary can read and reply/replay too.[5]

1. Client and Server perform full TLS handshake and interaction, adversary consumes messages originating from client
2. Adversary waits an appropriate amount of time for random to be removed from server cache(the key is that would normally be the same amount of time for server to pick a new random)
3. Adversary replays `ClientHello` assuming server will be happy to engage
4. Server returns certificate, which adversary doesn't really care about and accepts. Then the `ServerKeyExchange` takes place, and adversary has either diffie hellman parameter or enough to build master secret.
5. Handshake proceeds to `pms` and `msk` setting, and finished messages are performed.
6. Adversary is now able to read all previous (client-server) messages in plaintext at the application layer, and even produce productive messages to the server fully understanding what they entail.

2.3 Imagine a version of TLS without the H of the handshake message

The finished messages are the last set of messages sent before the connection is deemed secure and application data starts to stream between client and server. The `ClientFinished` and `ServerFinished` messages include the "H"(handshake).[1] This handshake denotes a hash of all of the data from all messages in the TLS handshake process, where either party uses this to verify that everything is correct on both ends, and communications can start. If either disagrees with any of the messages contained, the handshake fails.[1] The issue is that if an adversary positions itself as a middle-man, acting as a server to the client, and client to the server, the adversary can make it such that the client and server request the weakest possible ciphers without the other knowing.[1] Essentially, the adversary can make it seem like the server only supports those weak ciphers and vice-versa. And, since a handshake message is not included neither can see that the server and client actually did support stronger ciphers and that there was tampering. Essentially, the MiTM simply picks the weakest cipher-suite that both client and server support to perform this weakening.[4]
MiTM attack:

1. Adversary positions itself as man in the middle either by ARP poisoning or DNS spoofing.
2. Client sends `ClientHello`, adversary intercepts and discards.

3. Adversary sends the same `ClientHello` replacing the `CipherSuiteList` with the subset of ones it deems weak enough to crack.

4. The server replies with the `ServerHello` with its own `CipherSuiteList` to the client picking one of the weaker ciphers the adversary sent

5. At this point the attack is done, the client and server will not be able to realize that there was tampering. The client is left to presume the server only worked with that weaker cipher, while the server presumed the weaker cipher is the strongest the client could handle.

6. Finish messages go through and neither client nor server can detect the inconsistency of the `ClientHello`, and communications take place in a "secure fashion"

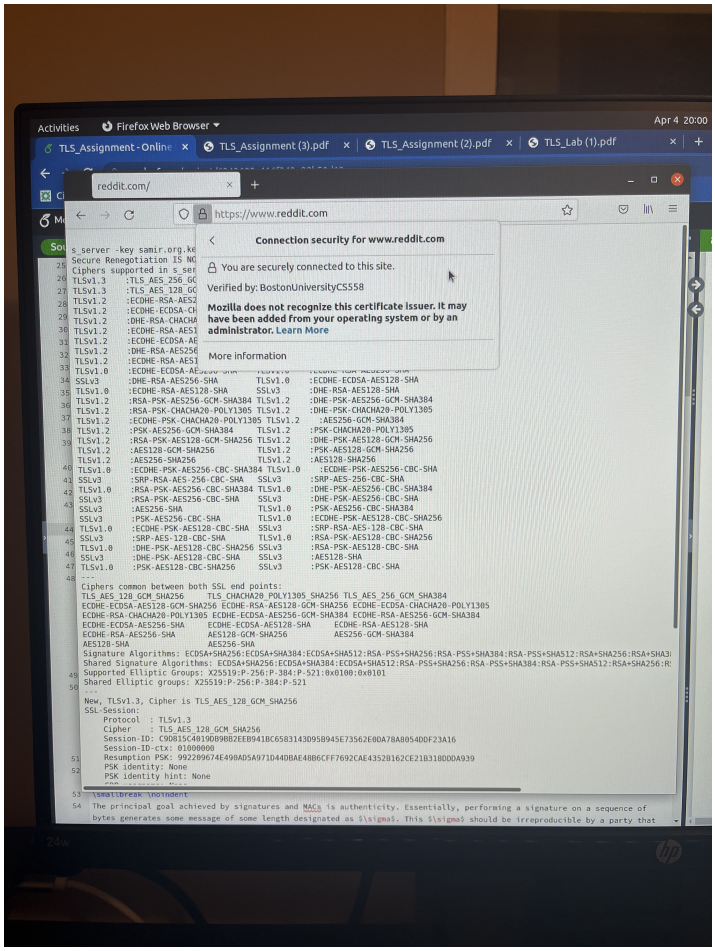
7. The adversary returns later to crack the cipher and is able to read all plaintext and decipher keys.

3 Tricking your Browser

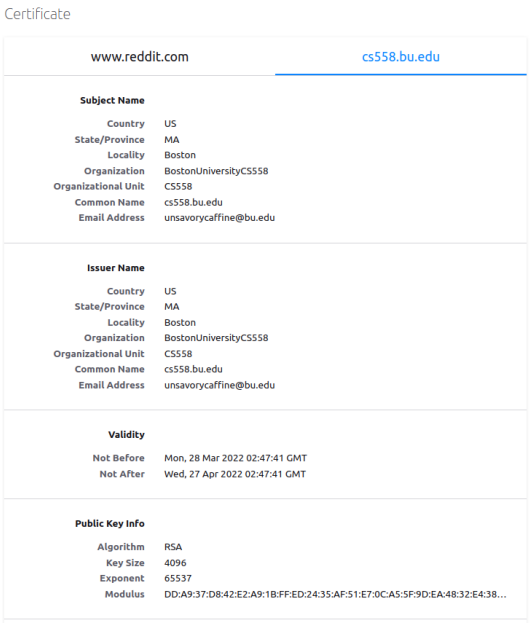
3.1 Diary

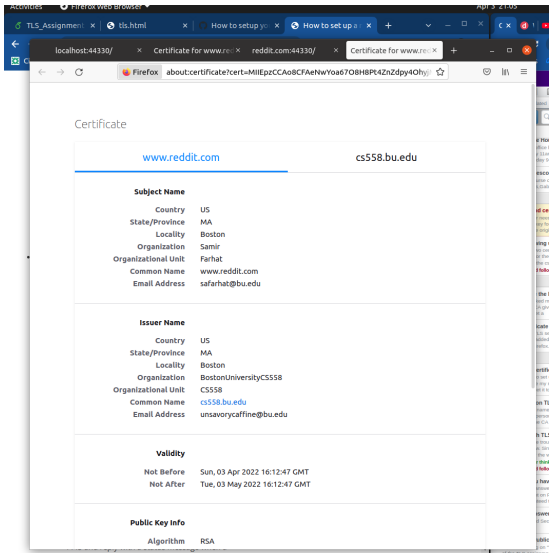
The tricking my browser portion of this work was a horrific experience, mostly due to my own incompetence. Things started out pretty smooth, I followed the guide step by step, generating the key, certificate signing request and my own certificate as normal. I then added the original CA certificate to my Firefox's list of certificates. Set up a proxy with my Ubuntu and moved on to the server setup part. Unfortunately the bud implementation was either deprecated or not functional, so I lost a couple of hours trying to get that to work before finally deciding to move on to openssl's `s_server` utility. I ran the command for that as below, and that's were several time loss issues occurred. First, I kept connecting as `localhost`, which after many hours trying to find a solution I found out I was meant to connect to the domain directly with the web address I specificed in the Common Name. Unfortunately this didn't resolve my issue either. After several more hours I finally decided to simply add the port number for the heck of it. And voila, it was working.

3.2 Successfull Connection



*Successfull connection taken with phone as Firefox closes lock pop-up when screenshotting normally.





*CA certificate and spoofed certificate respectively shows the chain starting at CS558 CA

3.3 Step-by-Step

The following step by step guides assumes a Ubuntu 20.04 OS and a relatively up-to-date linux kernel and bash shell. We will be impersonating `www.reddit.com`.

1. As a preliminary you should install openssl.

   ```
   sudo apt-get install openssl
   ```

2. Begin by saving the CA certificate and Private key to your local machine. For the sake of this example these will be named `ca.cert` and `private.key` respectively.

3. Feel free to perform a sanity check on the key.

   ```
   openssl rsa -in private.key  -noout -text
   openssl x509 -in ca.cert -noout -text
   ```

   If they run correctly and produce reasonable outputs you should be fine. The email "unsavorycaffine@bu.edu" should be present along with "MA", "Boston" and "CS558" plastered around.

4. Add the domain address to `/etc/hosts`.

   ```
   sudo vim /etc/hosts
   127.0.0.1    www.reddit.com
   ```

5. Generate a private key for your own domain's certificate.

```
openssl genrsa -out samir.org.key 2048
```

6. Generate a Certificate Signing Request(CSR) as below.

```
openssl req -new -key samir.org.key -out samir.org.csr

Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Boston
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Samir
Organizational Unit Name (eg, section) []:Farhat
Common Name (e.g. server FQDN or YOUR name) []:www.reddit.com
Email Address []:safarhat@bu.edu
A challenge password []:
An optional company name []:
```

7. Generate a self-signed certificate.

```
openssl x509 -req -in samir.org.csr -CA ca.cert -CAkey
             private.key -CAcreateserial -out samir.org.crt
```

8. Perform a sanity check ensuring the serial numbers and common name match.

```
cat ca.srl
openssl x509 -in samir.org.crt -noout -text
```

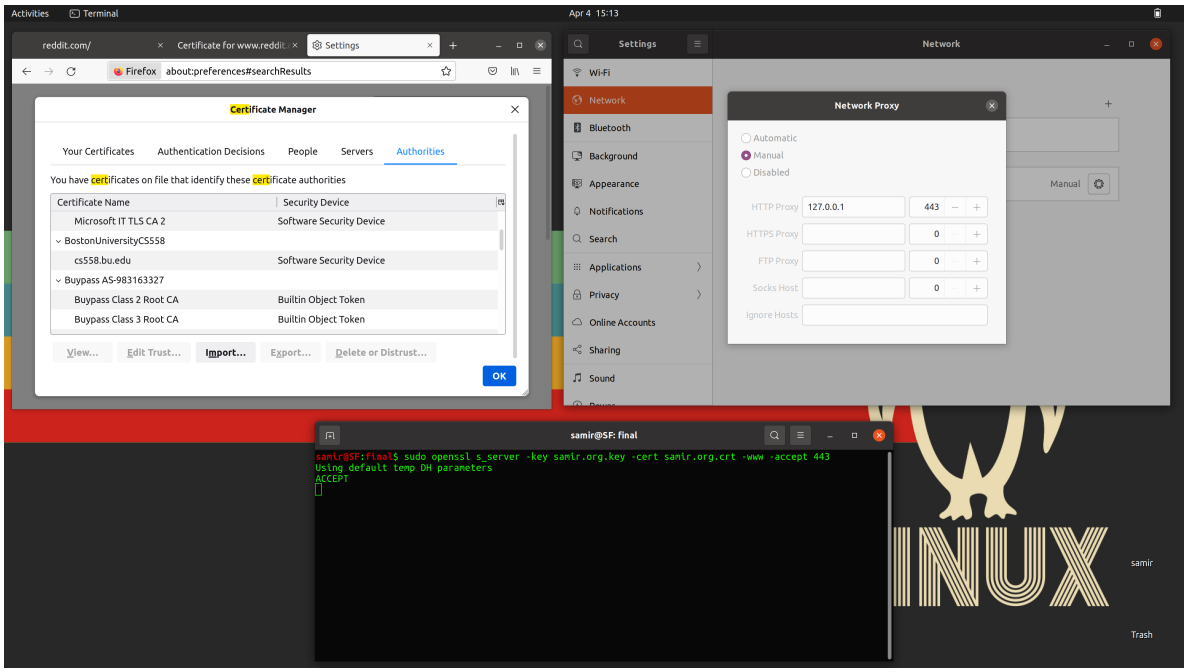
9. Now in order to trick out browser we want to generate a server, interestingly enough openssl has a functionality for this too.

```
sudo openssl s_server -key samir.org.key -cert samir.org.crt
                -accept 443 -www
```

10. Now add the ca.cert to list of authorities in Firefox. Do this by going to settings, entering "certifcates" in the search bar, clicking "view certificates". Go to the authorities tab and hit "import". Select the ca.cert file.

11. Now set up the proxy. Go to system settings->Network->Network Proxy. Select Manual and enter "127.0.0.1" for the IP and set the port to the one you made for the one you set on the openssl server. Our case is 443. That should do it. Type in the domain in Firefox and you should see your barebones server instead of reddit.

These last 3 steps are depicted below:



12. Go into your browser and simply type "www.reddit.com" you should see your bare-bones ssl server instead of Reddit.

Bibliography

[1] Dierks, Tim, and Eric Rescorla. "The transport layer security (TLS) protocol version 1.2." (2008): 5246.

[2] Rescorla, Eric, and Tim Dierks. "The transport layer security (TLS) protocol version 1.3." (2018).

[3] Davies, Joshua. Implementing SSL/TLS using cryptography and PKI. John Wiley and Sons, 2011.

[4] Böck, Hanno, et al. "Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS." 10th USENIX Workshop on Offensive Technologies (WOOT 16). 2016.

[5] Dutta, Avijit, Mridul Nandi, and Suprita Talnikar. "Beyond birthday bound secure MAC in faulty nonce model." Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Cham, 2019.

[6] Sikeridis, Dimitrios, Panos Kampanakis, and Michael Devetsikiotis. "Post-quantum authentication in TLS 1.3: a performance study." Cryptology ePrint Archive (2020).

[7] Holz, Ralph, et al. "The era of TLS 1.3: Measuring deployment and use with active and passive methods." arXiv preprint arXiv:1907.12762 (2019).

[8] van Thoor, Jules, Joeri de Ruiter, and Erik Poll. "Learning state machines of TLS 1.3 implementations." Bachelor thesis, Dept. Comput. Sci., Radboud Univ., Nijmegen, The Netherlands (2018).