Problem 1

1.

What is POODLE attack?

Poodle stands for Padding Oracle On Downgraded Legacy Encryption. It is a type of protocol attack carried out by downgrading the protocol's level to SSL3.0. The attacker, as a man in the middle, first tries to downgrade the TLS connection to SSLv3. This can be done by repeatedly dropping the session and eventually client will compromise with lower SSL version. Then, if the cipher suite uses RC4 or Block cipher in CBC mode, attacker can retrieve partial bytes of encrypted text and later on can get full plain text.

How to protect yourself (a client) from this attack?

Just disable SSL3.0. More effectively, disable any other versions of protocols that are susceptible to downgrade attacks.

2.

What is ssl striping attack? Explain how HTTP Strict Transport Security (HSTS) defends against the SSL Stripping attack.

SSL stands for Secure Sockets Layer. Back in the days, internet users are not used to type in "https". An adversary can effectively strip https requests into "http" by man in the middle. HSTS works as the following. First, the domain side is able to return HSTS header over HTTPS connection based on Trust-On-First-Use assumption. Secondly, the HSTS header add its domain to the browser's internal "preload list". Next time "http" is accidentally input for communication, the browser redirects to https internally before sending the request.

It seems like HSTS can store one bit of information (whether to use HTTPS or not) in a clients browser. Can you leverage this to create a super cookie to track a user even when the user is browsing your website?

HSTS, despite its original benign intention, is actually a super cookie. In fact, one has to explicitly clear his/her own HSTH record. "Clearing cookie" won't be effective. The server can leverage the bits information to identify a past website visitor. In detail, the server asks client to load http://bits[i].example.com, where the browser would internally establish "https" connection according to the past stored record. Moreover, ads. can look for relative HSTH records to learn from the client's past browsing history. It can then be used to proper advertising.

How to prevent this kind of attack?

Limit HSTS State to the Hostname, or the Top Level Domain + 1. Attackers set HSTS across a wide range of subdomains at once. This facilitates tracking. Thus, we revised our network stack to only permit HSTS state be set for the loaded hostname (e.g., "https://a.a.a.a.a.a.a.a.a.a.a.a.example.com"), or the Top Level Domain + 1 (TLD+1) (e.g., "https://example.com").

Ignore HSTS State for Subresource Requests to Blocked Domains. When a domain's cookie request have been blocked, we ignore the HSTS upgrade and use the original URL.

3.

What is a Tor Entry Guard Relay and how does it work?

Entry Guard Relays are the first relay of the entire tor circuit. Tor selects a fixed list of benign relays as entry guards, and rotates the clients' entry guard every 4-8 weeks. The original design over relay selection is uniformly distributed. Now relay selection is based on reliability and security. Usually, guard relays won't change frequently, otherwise the probability of a client being vulnerable at least once will increase due to birthday attack.

What is a Tor Bridge and how does it work?

Tor bridges are relays that are not listed publicly in order to circumvent censorships. The reason being for such design is that adversary censors can deny access on all publicly listed tor relays. As tor bridges are not publicly listed, censors don't know what IP address to block.

Problem 2

1. AS999 announce 10.10.220.0/23, 10.10.222.0/23

2. {10.10.220.0/23,{as999,as2,as1}}, {10.10.222.0/23,{as999,as2,as1}}

3. Loop prevention: Avoid specific routers from route selecting due to loop, and gather attention from other routers to create traffic congestions on targeted paths. Path prepending: adding own AS to leftmost side in order to make path less attractive from selection.

4. Pros: Prevent prefix hijacking. Save memory for IP routing table. Cons: Unfriendly to small ASs that provides only small range of IPs. Or even ASs that IPs are scattered, i.e. 512 IPs, 256 IPs in 9.10.220.0/25 and 256 IPs in 10.10.220.0/25.

Problem 3

1. From row 27 { s1 s2 s3 } -> { r4 } and row 41 { s1 s2 s4 } -> { r3 }, we are certain that r3 and r4 are recipients from s1. Observe that row 65 to row 91, { s1 s3 s4 } has never sent a single package to { r2 }. Therefore, a *strong assumption* could be deduced, r2 is not one of the recipients. If such assumption holds true, according to row 101 to row 109, { r2 } exists in all of the packages sent from {s1 s4 s5}. As a result, we can make another *strong assumption* that s5 only sends messages to r2. As for all messages sent from s5 is to r2, from row 56, { s1 s2 s5 } -> { r1 r2 }, we can derive that r1 is also one of the recipients from s1. For r3, we are really uncertain since there aren't any potential relationships observable among these data.
To sum up, we classify **r1, r3, r4 as absolute recipients**. **r2 is definitely not a receiver**. Whether **r5 is a recipient or not is uncertain**.

2. We can use a larger pool size for "b" such that we can create a more significant mix(larger entropy). Moreover, less information is provided for the fewer combination, 4 out of 5 as oppose to 3 out of 5, in this particular case. We can also produce meaningless packages and force all recipients receive meaningless packages whenever no connection is established. This is however based on the assumption that an observer cannot differentiate between garbage packets and meaningful packets.

Problem 4

1. If signatures σP(a, b), σQ(a, b) aren't encrypted, in the third step of this three way communication, a man in the middle adversary E can send σE(a, b) back to Q. In this case, Q thinks he/she is communicating with E, and P thinks he/she is communicating with Q. To be more specific, the participating E drops CertP and σP(a, b) that is meant to be sent to Q, and *attaches his/her own CertE and σE(a, b) to Q.*

2. Identity-misbinding attack can still be carried out towards Q. In the third step of this three way communication, a man in the middle adversary E can block messages that should be sent to Q, asks P to encrypt message σE(a, b), and send the returned Ek(σE(a, b)) along with CertE to Q.

3. As E knows information regarding P's public key, E can ask for certificate authentication from a *trusted* CA by showing P's public key along with E's identity. Eventually, E will be rewarded with CertE. As a man in the middle adversary, E is able to replace CertP with CertQ, and send CertE along with Ek(σP(a, b)) to Q. Since E and P share identical public keys, Q thinks he/she is talking to E according to the attached CertE.

4. Since we can suppose the hash function in the RSA signature scheme is an identity function,

$$\sigma P(a) = E(H(a)) = E(a)$$

where *"a"* is a pre-decided key. P is the targeted person we want to impersonate, and "*H(a)"* is the hash function that is same as an identity function.
We take advantage of the malleable property from RSA schemes. That is,

$$E(ab) = E(a) * E(b)$$

First, we request for sufficient amount of "*m*", i.e., "*m1, m2, …, mn"*.
Second, we ask P to sign on all messages, "*m1, m2, …, mn"*, and obtain "*σP(m1), σP(m2), …, σP(mn)".*
We select a secret "*x*" such that "*a = g^x"*.
If computational feasible, we can find factors for "*a*" among the *"m"*s, such that

$$a = m1^{(y1)}\ m2^{(y2)} \ldots mn^{(yn)}$$

As a result,

$$\sigma P(a) = \sigma P(m1)^{(y1)} * \sigma P(m2)^{(y2)} * \ldots * \sigma P(mn)^{(yn)}$$

As we have *"a", "σP(a)",* and *"certP",* we are capable of impersonating P.

Problem 5-1

How to run code:

1. Run code5-1.py with python3
2. After about 3 seconds long, the flag will appear at the end.

Solution:
Oracle attack by communicating with the server.

Explanation:
Observe that the "Check" function requires non-negligible computing power, and observe that the verification algorithm isn't buffed, we can exploit such attribute to check how close our secret is towards the standard answer. To be more specific, we send "1" to the server and obtain the total communication time. Since the length of secret doesn't match with the length of the standard answer, the communication is terminated early. We increase the number of bits sent to the server, i.e. "11", in an incremental matter, until we observe a significant time difference during the communication session, i.e. "1111111111". The next step is to enumerate through all the cases for this 10-bit long string, which in worst case requires 100 attempts. However, since the "Check" function also works iteratively starting from the first-bit, we can still exploit the time difference from the server's computing time. We then uncover the standard answer from "8111111111", "8611111111", …, "8604263255".

Flag: CNS{this_is_TIMMinY_attack!!!!!}

Problem 5-2

How to run code:

1. Run code5-2.py with python3
2. Let it run about 1-2 minutes long, the flag will appear at the end.

Solution:
Exploit bad seed settings used by Timmy's pseudorandom generator.
Build a table containing all MAC cases with one selected nonce, and perform birthday attack according to the table.

Explanation:
After playing with "Timmy2.py" several times, we observe that occasionally, Timmy announce reused nonce accidentally. Turns out that the seed used in this pseudorandom generator is flawed design. Firstly, Timmy renews his seed whenever a new communication session is established. This renders the randomness be dependent to the space of the seed. Running "findNonces.py" (*with Python version = 2.7*), we then discover that Timmy merely uses 128 different seeds. The seeds/ nonces results are stored inside "seedsAndNonces.txt" and "Nonces.py". The next step is to perform birthday attack by communicating with Timmy using a particular nonce. In "code5-2.py", we first gather all possible MACs of the following nonce pair, "mac(nonce[k], nonce[0]), for 0 <= "k" <= 128". Then, we select another nonce different to nonce[0], namely, nonce[1], and use nonce[1] to communicate with Timmy. Once Timmy

sends nonce[0] back, since “mac(nonce[1], nonce[0])” is precomputed inside a table, we pass the mutual authentication and obtain the flag.

Flag: CNS{>Be_4ware_0f_the_$33D<}

Problem 6-2

Why you can decrypt some of the messages in the captured packets, but not all of them?

Explanation:
Because only 2/5 of the packets are transferred under perfect forward security, ECDHE. Even though the long term key is exposed, short term keys cannot be derived. Those that can be decrypted uses RSA, which is not a perfect forward secured ciphersuite.

Problem 7-1

Solution:
A -> **N_A, id_A** -> B
B -> **N_A, N_B, id_A, id_B** -> KDC
KDC -> **E_A(k), M_A(id_B||N_A||N_B), E_B(k), M_B(id_A||N_A||N_B)** -> B
B -> **E_A(k), M_A(id_B||N_A||N_B)** -> A

Problem 7-2

How to run code:

1. Run code7-2.py with python3
2. The flag will appear at the end.

Solution: Man in the middle attack, replay attack.

Explanation:
A -> **N_A, id_A** -> HSU

B -> **N_B, id_B** -> HSU

HSU -> **N_A, N_B, id_A, id_B** -> KDC
KDC -> **E_A(k),** ***M_A(id_B||N_A||N_B),*** **E_B(k), M_B(id_A||N_A||N_B)** -> HSU

HSU -> **N_A, N_B, id_A, id_HSU** -> KDC
KDC -> ***E_A(k),*** **M_A(id_HSU||N_A||N_B),** **E_HSU(k),** **M_HSU(id_A||N_A||N_B)** -> HSU

HSU -> **E_A(k), M_A(id_B||N_A||N_B)** -> A
HSU solves for key from **E_HSU(k)** and obtains shared key.
HSU solves for the encrypted flag.

Flag: CNS{M4n_1n_Th3_Middl3_4nd_r3pl4y_4tt4ck_t0_K3y_Exp0sur3_Att4ck}