CNS, Spring 2020 HW1

B07902064 資工二 蔡銘軒

Handwriting

1. CIA

   - **Confidentiality:** Confidentiality requires information to be protected from unauthorized access and misuse. Communications during wars often require confidentiality. If the messages were to be leaked to the enemies, it would put the army in a very dangerous situation.
   - **Integrity:** Integrity requires information to be protected from unauthorized alteration. As an example, it violated the integrity requirement when a student in NTU hacked into the CEIBA platform and changed everyone's scores to 87.
   - **Availibility:** Availibility requires information or a system to be available to authorized users. As an example, it violated the availibility requirement when hackers in China launched a DDoS attack against Telegram during protests in Hong Kong.

2. Hash Function

   Let $H : X \to Y$, where $X = \sum^*$ and $Y = \sum^d$ for some $d \in \mathbb{N}$, be a hash function.

   - **One-wayness:** Assume $H(x) = y$ for some $x \in X$ and $y \in Y$. Given y, it is computationally infeasible to find x. One-wayness is required when a server hashes a user's password and stores it. If the hashed passwords were to be stolen by an attacker, lacking one-wayness would compromise the passwords and defeat the purpose of the hash function.
   - **Weak collision resistance:** Given $x \in X$, it is computationally infeasible to find $x' \in X$ such that $x' \neq x$ and $H(x) = H(x')$. Many services provide the hash values of the files that are open for download. If an attacker can easily create another malicious file that contains the same hash value as the original file, the users may not be able to tell whether the downloaded file is the intended one.
   - **Strong collision resistance:** It is computationally infeasible to find $(x, x') \in X \times X$, such that $x \neq x'$ and $H(x) = H(x')$. Consider a system which identifies users by the hash values of their information. If different members have different priviledges, a collisionn may cause two different people to be able to access each other's priviledge and information.

3. Semantic Security

   We denote the ordinary game as SS and the real/random game as RR.

   - $\Longleftarrow$ We prove that if the cipher is semantically secure in the ordinary sense, then it is real/random semantically secure.
     Let A be the adversary in SS. A can create another adversary B playing the RR game. We show that if B has non-negligible advantage, then so does A.
     The game is as follows:
     When B sends a message m to A, A sends (m_0, m_1) to its oracle, where $m_0 = m, m_1 \xleftarrow{R} M$. The oracle would give $E(pk, m_b)$ to A, A then forwards it to B and returns whatever B returns.
     We have $Pr[E_{SS}(0) = 1] = Pr[b \neq \hat{b} | b = 0]$ and $Pr[E_{SS}(1) = 1] = Pr[b = \hat{b} | b = 1]$. Since $b = 0$ and $b \neq \hat{b}$ are independent, we have $Pr[E_{SS}(0) = 1] = Pr[b \neq \hat{b}]$. Similarly, $Pr[E_{SS}(1) = 1] = Pr[b = \hat{b}]$.
     Then we have $|Pr[E_{SS}(0) = 1] - Pr[E_{SS}(1) = 1]| = |(1 - Pr[b = \hat{b}] - Pr[b = \hat{b}])| = 2|Pr[b = \hat{b}] - \frac{1}{2}|$

Capture The Flag

4. Simple Crypto

   Flag: `CNS{CRYPTO_1S_SO_$IMP13}`
   Explanation: The problem was solved with "code4.py".

- **warmup[full-auto]:** Simply copy and paste the given message. Done automatically by the program.
- **round 1[full-auto]:** The message was encrypted with ROT13. Decrypted automatically by the program.
- **round 2[half-auto]:** The message was encrypted with Caesar Cipher. However, I found no particular shift used in this round, so the program would compute all the 26 shifts and prompted the user for the correct shift used in this round.
- **round 3[half-auto]:** Identical to round2.
- **round 4[half-auto]:** The message was encrypted with Substitution Cipher. The program deduced the substitution from $c1$ and $m1$. However, some substitutions needed to decrypt $c2$ might not be found in $c1$ and $m1$. In this case, the program would output the decrypted part and replaced the unknown characters with "?", and prompted the user to fill in the "?". Usually the decrypted part provided enough information for a user to deduce the missing words.
- **round5[full-auto]:** The encrypted message was a rearrangement of the original message. Let C be the encrypted message, M be the decrypted message, L be the length of M and C, and $C[i]$ be the *i-th* character in C. The encryption followed the following rule:

  $$M = C[0]C[t \ mod \ L]C[2t \ mod \ L]C[3t \ mod \ L] \dots C[(L-1)t \ mod \ L] \quad \text{for some } t \in 1, 2, \dots, L-1$$

  The program found the t used in the round and decrypted the message automatically.
- **flag[manual]:** The flag was encoded in Base64. I used an online tool to decode the flag.

5. Find The Secret

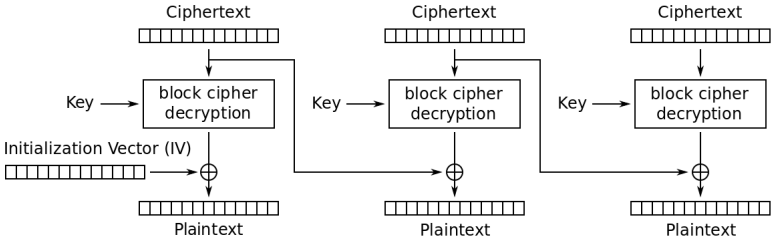
Flag: `CNS{V3rIfi4BLe!!}`
Explanation: The problem was solved automatically by "code5.py"
From the information given in the problem statement, we have $g^q \equiv 1 \ mod \ p$. It follows that

$$g^{D_1} \equiv g^{a_0+a_1+a_2 \ mod \ q} \equiv g^{a_0} g^{a_1} g^{a_2} \equiv c_0 \cdot c_1 \cdot c_2 \ mod \ p$$
$$g^{D_2} \equiv g^{a_0+2a_1+4a_2 \ mod \ q} \equiv g^{a_0} (g^{a_1})^2 (g^{a_2})^4 \equiv c_0 \cdot c_1^2 \cdot c_2^4 \ mod \ p$$
$$g^{D_3} \equiv g^{a_0+3a_1+9a_2 \ mod \ q} \equiv g^{a_0} (g^{a_1})^3 (g^{a_2})^9 \equiv c_0 \cdot c_1^3 \cdot c_2^9 \ mod \ p$$

After checking each condition for D_1, D_2, D_3, I filtered out all the fake D_1, D_2, D_3, luckily. I then reconstructed the function, denoted by F, with the real D_1, D_2, D_3 and the lagrange interpolation code found on Wikipedia. Now that $a_0 = F(0)$, I translated it from decimal to binary, and used "UTF-8" to decode the binary string. The decoded string turned out to be the flag.

6. Cute Baby Cat

Flag 1: `CNS{A_lonely_cat_wan||isvip:0||isadmin:0||desc:ts_friends.Meow~meow~(=^._.^=)}`
Explanation: The flag can be obtained by performing the padding oracle attack.



There are three blocks in the picture above. Let c_i, m_i, d_i be the cipher text, plain text and the output of the cipher of the *i-th* block, respectively.
If we focus on the decryption of the second block, we can see that m_2 is obtained by performing $Xor(c_1, d_2)$. If we want to know the last byte of m_2, denoted by s, we can simply enumerate all the 256 possibilities.
Let s' be our current guess, we modify c_1 by changing its last byte to $Xor(1, \text{last byte of } c_1, s')$. Let

the modified c_1 be c_1', we check if $Xor(c_1', d_2)$ gives a correct padding for m_2. All the other bytes can be decrypted similarly.

Flag 2: `CNS{(=^.x.^=)W15H YOU H4V3 FUN}`
Explanation: Using the same concept in padding oracle attack, we can manipulate the plain text at will by modifying the cipher text of the previous block, denote by c_p.
Here we try to forge `isvip:1`. To do so, we need to modify the corresponding byte in c_p. The problem is that, after modifying c_p, it cannot be decoded successfully. We need to modify other bytes to remedy this situation.
We can only modify the first four bytes, as they correspond to the "name" section, which can be arbitrary, while the other bytes cannot be changed in order to manipulate the plain text in the latter block. A legitimate cipher can be obtained by enumerating the first four bytes, and then we can login to get the flag.

Flag 3: `CNS{B1t_FL1p_4T7Ack_15_Tr1ckY.a_cut_cat.gif}`
Explanation: The idea is similar to the previous problem. I will explain it using the picture above. Let the second block and the third block correspond to where `isvip` and `isadmin` are, respectively. In order to forge `isadmin:1`, we must modify the corresponding byte in c_2. However, this disturbs the decryption of c_2. Again, if we know the plain text of a block (or the output of the cipher), we can manipulate the plain text by modifying the cipher text of the previous block. So we use padding oracle attack to get the output of the cipher after changing c_2 and modify the bytes in the first block to make `isvip:1` appear in the plain text of the second block. Now we are back to the situation where the first block cannot be decoded successfully. We again enumerate the first four bytes until we get a legitimate token.

7. Resourceful Secret Agent

Flag 1: `CNS{T3xtb0ok_RSA_1s_uNSaF3}`
Explanation: To solve this problem, I used the malleable property of RSA.
We have $E(pk, x) \cdot E(pk, y) = E(pk, x \cdot y)$, where pk is the key we don't know, x is the message we want, and y is any number we get to choose. I chose $y = 2$, and obtained $n, E(pk, y)$ from the server, and asked the server to decrypt $E(pk, x \cdot y)$. After obtaining $x \cdot y$, I simply divided it by y to recover x.
Flag 2: `CNS{eNCrYp7_Y0uR_s3cr3T_w1sely}`
Explanation: The **mac** is generated using OFB mode. So the message is xored with a byte stream to produce **mac**. As we have both **mac** and the message (obtained in the previous problem), we can easily recover the byte stream. We can now change the suffix of the message from "**mac**" to "**getflag**", and xor it with the byte stream to generate the legitimate **mac**. Finally we format the string properly, send it to the server, and receive the flag.

8. Wired Equivalent Privacy

Flag 1: `CNS{r3us3_K3Y_br3ak_OTP}`
Explanation: I first obtained the key (explained in the next problem) to generate the byte stream used to encrypt each message. I captured tens of thousands of packets and simply recovered all the packets with the key. Finally I looked for the keyword "**CNS**" and found the flag.
Flag 2: `CNS{D0_N0T_Us3_WEP_4nym0r3}`
Explanation: I performed the "**FMS attack**" to recover the key.
The basic concept of the attack is that, we first look for **iv** of the form $(A+3, N-1 = 255, V)$, where A is the number of bytes we know in the key (we start with 0), and V could be any number.
In the following pictures, the first row is the sesson key K (which is **iv** + key), and the second row is the status of the **S-box** S.
The pictures for the first and second round of the **KSA** is as follows:

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	1	2			0	
i_0					j_0	

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	0	2			1	
	i_1				j_1	

All the rounds later can be calculated in the same manner. We continue until we encounter the $A+3$ byte, which we don't know. We first check if either $S[0]$ and $S[1]$ has been changed. If not, we know j will be added $S_{A+2}[i_{A+3}] + K[A+3]$ and then $S[i]$ will be swapped with $S[j]$. The status would become

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	0	$S[2]$			$S_{A+3}[A+3]$	
					i_{A+3}	

At this point, all we need is j_{A+3} to compute $K[A+3]$.
I collected around 20000 packets to perform the attack, and successfully recovered the key.