MIDN Andres

HW06/07

1. (5 pts) Read the course notes on Modes of Operation, MAC, and HMAC. Honestly declare one of the following:
    a. (0 pt) I didn't read the notes.
    b. (2 pts) I just skimmed the notes.
    c. (3 pts) I read the notes, but I skipped a couple of difficult topics.
    d. (5 pts) I read the notes very carefully and thoroughly.

D

2. (5 pts) Solve the quiz candidate problems on MAC, HMAC, and authenticated encryption. Honestly declare one of the following:
    a. (0 pt) I did nothing.
    b. (2 pts) I looked at the solutions, but didn't attempt to solve the problems on my own. (Recall you should get help from your instructor, if you don't understand the course material.)
    c. (4 pts) I tried to solve **some** of the problems on my own, and then checked the solutions.
    d. (5 pts) I tried to solve **all** problems on my own, and then checked the solutions.

D

3.

a. Explain why you obtain such a decryption result. Try to explain how different the two files and explain why you have such a difference.

   In this problem we changed one byte of information in the ciphertext, the result was a decrypted message that only corresponds to about half of the actual message. In CBC encryption and decryption depends on each succeeding block of the message. In this case, the byte we modified after we encrypted was the 13th-byte position out of 29 bytes. So when we decrypted it, we start at the end of the message (the 29th-byte) and it works correctly until we hit the byte we modified, then there is no way to decrypt the rest of the message because it depended on that 13th-byte being correct.

b. Try to decrypt the cbc2.out with the same key. Compare the decryption message with the contents in the original M.txt.

   (5 pts) Explain why you obtain such a decryption result. Hint: Recall how the CBC mode works.

   In this case, we modified byte-14 of the ciphertext and the result was a bad decrypt, an error that did not even allow us to attempt to recover the message. As I mentioned before CBC encryption and decryption depend on each succeeding block of the message, but it also depends on its IV and correct padding. Because we modified byte 14, when we got to the decryption of the 14th byte, the decryption algorithm determined that it was incorrect padding and thus the decryption failed.

c. (5 pts) Based on the results above, come up with an attack to figure out the padding length of a given ciphertext (encrypted with the CBC), if you have access to a magical black-box telling you only whether a ciphertext is valid or not (i.e., padding error/oracle).

In order to figure out an attack to determine the padding length of a given ciphertext, we would need to first obtain the ciphertext. In our use of CBC, we know that the length of one block of a message corresponds to 16 bytes. Thus, we have 16 possibilities of what the padding length could be. We modify the ciphertext by replacing 1 byte of data out of 16 and send it to the decryption algorithm. We know if the decryption goes through, we are modifying the message. However, if we receive an error, we know that we are modifying the padding, thus we should start from the beginning index and work our way up until we get a bad decrypt. For example, if we modify byte 14 and we receive an error, then we know that the length of the padding is 16-14= 2 -> (02).

d. (10 pts EXTRA CREDIT) Based on the above observation, come up with an attack that can use the padding oracle to determine the exact contents of the message. Hint: the padding oracle will tell you whether the padding is correct or not. You can use this to tweak a ciphertext and check whether your tweak matches a certain value...

Once we know the padding byte value as determined in the problem above, we can decipher what the actual decrypted value is. All we have to do XOR it with the previous ciphertext. Using this technique, we can work our way backward through the entire block until every byte of the ciphertext value is cracked, thus getting the decrypted value one at a time.

4. (20 pts: 50 minutes) Recall the security notion (EU-CMA) of message authentication code. Given a MAC scheme, if an adversary forges a pair of a new message and a tag that passes a verification algorithm, then the MAC scheme is not secure. In this problem, you are supposed to show insecurity of a few MAC schemes.

a. Your Task: Show the above MAC scheme is insecure by doing the following. Create a newly forged message-tag pair (Mforge1.bin, Tforge1.bin) that will pass verify1. Explain how you created those files. If necessary, give a python file that you wrote to create those forged files. Show that your forgery is successful by executing the following and store the results in log1.txt.

Essentially what I was able to do is modify the message such that it is in a different order, but of the same content as the original message and utilize the same tag to verify the message. It will return valid but it is not the original message intended by the sender. This is made possible because the first 16 bits go through ECB encryption, as does the last 16 bits, then they are XORed together to create the tag. So we can modify the message such that it is in a different order and it will create the same tag. You could also erase the message by XORing the message and the tag with itself, giving you all 0's. This will produce a valid message tag pair, but none of the intended data will get to the recipient, I will attach my simple python script that produced these values for my first scenario.

b. Your Task: Show the above MAC scheme is insecure by doing the following. Create a newly forged message-tag pair (Mforge2.bin, Tforge2.bin) that will pass verify2. Explain how you created those files. If necessary, give a python file that you wrote to create those forged files. Show that your forgery is successful by executing the following and store the results in log1.txt.

We can use a splicing attack: Given (M1,M2),T), we can forge a legitimate pair of a message and a tag as follows: ((M1,M2,M1$\oplus$T,M2),T). I created a python program that extracted T and put it into its

original form. From there I created a byte array that had a length of 64 bytes. I filled the first 32 bytes with the contents of M.txt. Then I took the first 16 bytes from M.txt and XORed it with the reconstructed T tag, which was bytes 32-48. Then I appended the last 16 bytes of M.txt to give me the full 64 bytes. This message is clearly different from the original message, however, the tag was validated. I will attach my simple python script that produced these values.

5. (2 pts: 5 minutes) What is SHA-256 hash of usna.bmp? The following hexdump shows the SHA-256 hash of the file; fill out the blank.

00000000 ce 1e 88 68 24 75 e3 f1 64 19 6c 63 60 f4 e1 71 |...h$u..d.lc`..q|

00000010 fb 6c e2 cc 8d f8 08 e8 35 42 8f 9e 62 08 29 35 |.l.....5B..b.)5|

Explain how you got the answer:

I typed in the terminal " openssl dgst -sha256 usna.bmp" and got: SHA256(usna.bmp)= ce1e88682475e3f164196c6360f4e171fb6ce2cc8df808e835428f9e62082935