CS177: Computer Security
Prof. Stefano Tessaro

UC Santa Barbara
Fall 2018

Homework 2

Posted: Wednesday, October 10, 2018 – 11:59pm
Due: Friday, October 19, 2018 – 11:59pm (gradescope)

General rule (reminder!): Justify *all answers* in detail. Do not use sources other than the classroom slides to solve *any* of the task.

Task 1 – When IVs Collide (6 points)

Consider the following 32-byte randomized counter-mode ciphertexts (based on AES), produced using the same secret key (which is however unknown to you):

$C_1 =$ `FFBC1ADC607ACDDEAE7D837FA8123A3AE9F1B9C17D3281EE529B2FCD8ABBCA4D`
$C_2 =$ `FFBC1ADC607ACDDEAE7D837FA8123A3AF9F1B9C17D3281EE529B2FCD8ABBCA4D`

a) [Points: 2] Which information can you infer about the plaintexts encrypted by both ciphertexts? Why does this not contradict semantic security?

b) [Points: 2] Suggest a scenario where learning the information from **a)** can be problematic.

c) [Points: 2] Imagine we use randomized counter-mode encryption with Triple DES (3DES) instead of AES: the block length is now only 64 bits = 8 bytes. Why can this be a problem, independently of how secure 3DES actually is?[1]

Task 2 – CBC Integrity (4 points)

Consider the following 32-byte CBC ciphertext (based on AES) using PKCS#7 padding, produced with a secret key unknown to you.

`FFBC1ADC607ACDDEAE7D837FA8123A3A9CFDD83A9CD55F15A8CD7F8CFA32A67B`

We also learnt the ciphertext encrypts a plaintext of length 8 bytes.

a) [Points: 2] Suggest a modification of the above ciphertext which will certainly decrypt to a valid plaintext.

b) [Points: 2] Suggest a modification of the above ciphertext which *will likely not* decrypt to a valid plaintext.

[1] There are no serious attacks against 3DES (unlike plain DES), so the block cipher security itself is not a problem.

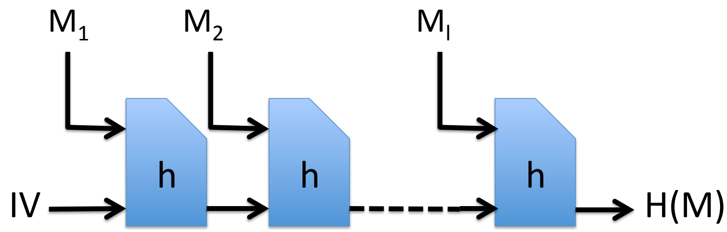

Figure 1: Diagram of a hash function construction following the Merkle-Damgård (MD) paradigm using the compression function h.

Task 3 – Integrity and MACs (6 points)

Hash functions like MD5, SHA-1, and SHA-256 are built from a (very efficient) *compression* function $h : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$. To compute $H(M)$, first, the message M is padded into b-bit blocks $M_1, \ldots, M_\ell$ as in Task 2. Then, the hash function outputs $H(M) = H_\ell$, where (for a given fixed initialization value IV)

$$H_0 = IV \ , \quad H_i = h(H_{i-1}, M_i) \text{ for all } i = 1, \ldots, \ell \ .$$

This construction approach is known as the Merkle-Damgård (MD) paradigm, and is illustrated in Figure 1.

We now build a message-authentication code $\mathrm{MAC}_K(M) = H(K \parallel M)$ from a hash function H, where the key K is a b-bit string, and $M \in \{0,1\}^*$ is an arbitrarily long message. (Here, $\parallel$ denotes string concatenation.)

a) [Points: 4] Show that MAC does *not* satisfy unforgeability if H follows the MD paradigm, i.e., given $(M, T = \mathrm{MAC}_K(M))$ for an unknown secret key K and a known message M, show that it is possible to efficiently find $M' \neq M$ and T' such that $\mathrm{MAC}_K(M') = T'$.

Hint: Show first that (regardless of what h is) one can always compute from $H(M)$ (using h) the hash $H(M')$ for a message M' related to (yet different from) M.

b) [Points: 2] Why is this attack not possible with HMAC?

Task 4 – Encrypt-and-MAC (6 points)

We consider Encrypt-and-Mac (E&M) as defined in class. In particular, data is encrypted using counter-mode encryption with AES, and then, to get the final ciphertext, we append to the counter-mode ciphertext a MAC of the *data* using HMAC with SHA-256.
Imagine that you intercept the following two ciphertexts C_1 and C_2, using the same secret

key for both (which is however unknown to you):

$$
\begin{aligned}
C_1 &= \text{43371380524753188fd571d8622ae61f64ccb551d9b348119adbc4} \\
&\ \text{10cbdef77cf180f7529d0da0c6f0a4fdb28a3d56e2105c7eb4b13b8c} \\
&\ \text{4cd9001523ba1e55dc2cd5608e84c093cd21d1126ddac1e7b2a5e9} \\
C_2 &= \text{853b56f79857359cad582eb6e6cb1a23b9a08d1c32e8638da80671} \\
&\ \text{44b9d781795a0a79496ca15ffe8865408aa83194df66d87eb4b13b} \\
&\ \text{8c4cd9001523ba1e55dc2cd5608e84c093cd21d1126ddac1e7b2a5e9}
\end{aligned}
$$

a) **[Points: 2]** What can you infer about the plaintexts encrypted by the two ciphertexts above? Justify your answer!

b) **[Points: 2]** Can E&M ever be semantically secure? Explain your answer!

c) **[Points: 2]** Does E&M satisfy integrity?

Task 5 – Padding-Oracle Attacks (18 points)

In class, we have seen an example of a padding-oracle attack which recovers one plaintext byte from a ciphertext encrypted with CBC encryption using PKCS#7 padding. The attack only needs to make so-called *validity checks*, each telling us only whether the padding inside the encryption is correct or not. We want now to elaborate on this attack.

a) **[Points: 2]** Consider the scenario from the class slide, where we want to recover the last byte of the last plaintext block (which may or may not be validly padded). Show that in general there may be two values X_1 and X_2 such that xoring $X_1 \oplus 0x01$ and $X_2 \oplus 0x01$ to the last byte of the second-last block leads to correct decryption.

Hint: What if the second-to-last byte of the last (plaintext) block has value $0x02$ and the last one has value $0x08$?

b) **[Points: 2]** In case both X_1 and X_2 lead to decryption, show that with one additional validity check we can determine which one of the two is the actual value.

c) **[Points: 4]** Explain how to extend the padding-oracle attack presented in class to recover the *entire message* M given its CBC encryption C. How many validity checks does your attack need?

d) **[Points: 10]** We now want to implement the padding oracle attack from **c)** against CBC. To this end, we provide `oracle.py`[2] which contains a function `PadOracle` which takes as argument a string (whose length must be a multiple of 16 bytes) and checks whether it encrypts a correctly padded message, for a hard-coded fixed key. In particular, it returns either `True` or `False` to indicate whether the padding is valid or not.

Extend `oracle.py` into a Python program that decrypts any given ciphertext (in a file whose name is passed as an argument) encrypted under the hard-coded key by *only using calls to* `PadOracle`.

[2] from https://www.cs.ucsb.edu/~tessaro/cs177/hw/oracle.py

Note in particular the following:

- `oracle.py` is meant to work with Python 2.7 on the CSIL cluster. So try to stick with that.
- You can test your implementation on two sample ciphertexts encrypted with the hard-coded key, available at `https://www.cs.ucsb.edu/~tessaro/cs177/hw/1.ctxt` and `https://www.cs.ucsb.edu/~tessaro/cs177/hw/2.ctxt`. Their correct decryption will result in English plaintexts with clearly recognizable structure.
- Only edit `oracle.py` in the designated area in the file (check out the comments). If run on a valid ciphertext, the latter will be in the variable `ctext`.
- The key is visible in `oracle.py`, but you should stick to the rules and *not* decrypt directly using it, but only indirectly using `PadOracle`.
- We will post further instructions and clarifications on Piazza whenever necessary, so check this out regularly. In particular, we will give some further hints on manipulating strings.