Homework 2

Posted: Thursday, April 7, 2016 – 11:59pm
Due: Monday, April 18, 2016 – 5pm (HFH 2108) or 12:15pm (in class)

Task 1 – Secret-Key Encryption  (6 points)

a) [Points: 2] Assume that you are given the ciphertext $C$ encrypting a message $M$ (which we expect to be English text) using a symmetric encryption algorithm with key length 128 bits (e.g., counter-mode or CBC encryption using AES with 128-bit keys).

What is the cost (in terms of evaluations of the underlying block cipher) of a brute-force key-search to recover a random secret key?

b) [Points: 2] Give an example of a property of the plaintext which is not hidden by semantic security.

c) [Points: 2] Does counter mode (CTR) encryption guarantee integrity?

Task 2 – Integrity and MACs  (8 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 $MAC_K(M) = H(K || M)$ 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, $||$ denotes string concatenation.)

a) [Points: 5] Show that MAC does not satisfy unforgeability if $H$ follows the MD paradigm, i.e., given $(M, T = MAC_K(M))$ 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 $MAC_K(M') = T'$.

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: 3] Why is this attack not possible with HMAC?
Figure 1: Diagram of a hash function construction following the Merkle-Damgård (MD) paradigm using the compression function $h$.

**Task 3 – Padding-Oracle Attacks** (21 points)

In class, we have seen an example of a padding-oracle attack which recovers one plaintext byte from a ciphertext encrypted with CTR 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] In class, we have assumed that the second-to-last byte of the block has value larger than $16 = 0x10$. Show that when this assumption is not true, then there may be two values $z'_1$ and $z'_2$ that lead to successful decryption.

**Hint:** What if the second-to-last byte of the block has value $0x02$ and the last one has value $0x08$. Which possible values of $z'$ lead to correct decryption?

**b)** [Points: 2] In case both $z'_1$ and $z'_2$ lead to decryption, show that with one additional validity check we can determine whether $z = z'_1$ or $z = z'_2$.

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

Also, explain how and whether your attack can be applied to CBC mode.

**d)** [Points: 13] We now want to implement the padding oracle attack from c against CBC. To this end, we provide `oracle.py`\(^1\) 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`.

Note in particular the following:

---

\(^1\)from https://www.cs.ucsb.edu/~tessaro/cs177/hw/oracle.py
- *oracle.py* is meant to work with Python 2.7 on the CSIL cluster. Unfortunately, CSIL does not seem to support the pycrypto library on Python3. 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](https://www.cs.ucsb.edu/~tessaro/cs177/hw/1.ctxt) and [https://www.cs.ucsb.edu/~tessaro/cs177/hw/2.ctxt](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`.

- Submit your solution using turnin. Use

  ```
turnin hw2@cs177 oracle.py
  ```

- 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.