Homework 2: Hash Functions, MACs and Length Extension and Padding Oracle attacks

Submission policy. Submit your answers on Blackboard by 11:59pm on **Friday**, March 8, 2019. Late submissions will be penalized according to course policy. Your need to TYPESET your answers and your submission MUST include the following information:

1. A pdf file named as **LastnameHW2.pdf** with with all the answers to the theoretical problems and theoretical parts of the Project Part. At the end of the .PDF you must include a list of references used (online material, course nodes, textbooks, wikipedia, etc.)
2. A compressed file including all the files needed for the Project part named as **LastnameHW2.zip**.
3. Your name should be included in EVERY single file submitted.

Administration. This assignment will be graded and administered by Panagiotis Chatzigiannis.

Structure. Every ISA 656 HW consists of two parts. The theory part and the practical part. HW2 accounts for 8% of your final grade.

Exercise 1. [18 points] CryptoBitPay is a micropayments company that needs to process millions of payments per second. The most expensive computation of each payment is that of a digital signature and they decide to decrease the overall cost of signing by using an alternative scheme. Instead of signing every message independently, they sign groups of 10 messages $m_1, \ldots, m_{10}$ together as follows:

- First compute $x = H(H(m_1)||H(m_2)|| \ldots H(m_{10}))$, where $H()$ is a collision resistant hash function and $||$ stands for string concatenation.
- Then, instead of signing each message separately, sign only x. Thus the signature s is equal to $s = Sign(sk, x)$ where sk is the secret key of the company.

The main idea of CryptoBitPay is that one signature in x is as good as 10 signatures on the 10 messages. Your goal in this problem is to evaluate the idea of CryptoBitPay.

1. **[5 points]** The value that is being signed with the new scheme is x. Assume that the company wants to send one of the messages m_i for $0 \leq i \leq 10$ to a customer and convince him that it was signed *without revealing the rest of the messages m include in x.* They decide to send m_i, x, s (assume that the customer knows the public key of the company pk). Explain why this information is not enough for the customer to verify that the message m_i was signed. What else would you send? How exactly would verification work on the customer side?

2. **[5 points]** Explain why it is not possible for an adversary to trick a customer that a message x' was signed when it actually wasn't? (Just explain the high level intuition).

3. **[4 points]** If CryptoBitPay's server takes 0.1s to compute one signature let's see how much is the company saving with new scheme. How many messages can now be signed in one second? (assume hashing comes for free).

4. **[4 points]** How many bytes of additional data need to be sent to the customer with each message m_i, in total? Assume that each signature is 32 bytes long and the hash function $H()$ outputs 32-byte hash digests. (Ignore the size of the specific message which in practice might be pretty large.)

5. **[BONUS points: 6 points]** You now have to come up with an improvement of CryptoBitPay's scheme so that the company's server can sign up to 10,000,000 messages per second, but at the same time the amount of additional data send to the client to verify that m_i was signed is les that 1000 bytes. How would you change the scheme? How many bytes of additional data need to be sent to the customer with each message m_i?

   (Simply including 1000 messages in the original scheme does not work since the amount of additional data exceeds by far the 1000 bytes limit.)

Exercise 2. [12 points] Consider the following public-key protocol for two entities A and B. The protocol assumes that A and B know each other's public key K_A and K_B, respectively. The protocol aims to establish a shared secret key between A and B.

1. A chooses a nonce N_a and encrypts (N_a, A) with the public key K_B of B, where A is the identifier of A. The ciphertext C_1 is sent to B.

2. B uses its private key S_B to decrypt the received ciphertext C_1, and obtains N_a. Also, B chooses a nonce N_b and encrypts (N_a, N_b) with the public key K_A of A. The ciphertext C_2 is sent to A.

3. A uses its private key S_A to decrypt the received ciphertext C_2, and obtains N_b. Also, A encrypts N_b with the public key K_B of B, and sends the ciphertext C_3 to B.

4. A and B each computes $k = \text{XOR}(N_a, N_b)$, which is used as the shared secret key. (They encrypt their communications with k in a symmetric key encryption scheme.)

Please answer the following questions.

1. **[3 points]** What is the purpose of ciphertext C_3 in the third step of the protocol?

2. **[5 points]** The protocol presented above has a security vulnerability that is similar to a man-in-the-middle attack. The protocol allows a third party P to impersonate A to communicate with B. Thus, at the end of the attack, B thinks that he is talking to A, but he is actually talking to P.

This can happen when A tries to communicate with P (for some legitimate request) as follows. A chooses a nonce N_a and encrypts (N_a, A) with the public key K_P of P. P then impersonates A to start the above protocol with B.

Please describe the rest of the attack by showing step-by-step the messages exchanged among A, B, and the attacker P. We assume that the public key of anyone is known to the public. The attacker P has the following capabilities: eavesdropping on the communication channel and intercepting ciphertext, participating in the protocol (initiating and carrying out communications), and sending messages with forged IP source address for impersonation.

3. **[5 points]** How would you change the second step of the protocol to fix the problem?

Programming assignment - Part A: Hash Functions

In this project, you will investigate vulnerabilities in widely used cryptographic hash functions, including length-extension attacks and collision vulnerabilities. In Part 1, we will guide you through attacking the authentication capability of an imaginary server API. The attack will exploit the length-extension vulnerability of hash functions in the MD5 and SHA family. In Part 2, you will use a cutting-edge tool to generate different messages with the same MD5 hash value (collisions). You will then investigate how that capability can be exploited to conceal malicious behavior in software.

Objectives:

- Understand how to apply basic cryptographic integrity primitives.
- Investigate how cryptographic failures can compromise the security of applications.
- Appreciate why you should use HMAC-SHA256 as a substitute for common hash functions.

Part A.1. Length Extension

In most applications, you should use MACs such as HMAC-SHA256 instead of plain cryptographic hash functions (e.g. MD5, SHA-1, or SHA-256), because hashes fail to match our intuitive security expectations. What we really want is something that behaves like a pseudorandom function, which HMACs seem to approximate and hash functions do not.

One difference between hash functions and pseudorandom functions is that many hashes are subject to *length extension*. All the hash functions we've discussed use a design called the Merkle-Damgård construction. Each is built around a *compression function* f and maintains an internal state s, which is initialized to a fixed constant. Messages are processed in fixed-sized blocks by applying the compression function to the current state and current block to compute an updated internal state, i.e. $s_{i+1} = f(s_i, b_i)$. The result of the final application of the compression function becomes the output of the hash function.

A consequence of this design is that if we know the hash of an n-block message, we can find the hash of longer messages by applying the compression function for each block $b_{n+1}, b_{n+2}, \ldots$ that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

1.1 Experiment with Length Extension in Python

This component is intended to introduce length extension and familiarize you with the Python MD5 module we will be using; you will not need to submit anything for it.
To experiment with the idea of length extension attacks, we'll use a Python implementation of the MD5 hash function, though SHA-1 and SHA-256 are vulnerable to length extension in the same way. You can download the `pymd5` module here (click to open) and learn how to use it by running `$ pydoc pymd5` [1]. To follow along with these examples, run Python in interactive mode (`$ python -i`) and run the command `from pymd5 import md5, padding`.

Consider the string "Use HMAC, not hashes". We can compute its MD5 hash by running:

```
m = "Use HMAC, not hashes"
h = md5()
h.update(m)
print h.hexdigest()
```

or, more compactly, `print md5(m).hexdigest()`. The output should be:

```
3ecc68efa1871751ea9b0b1a5b25004d
```

MD5 processes messages in 512-bit blocks, so, internally, the hash function pads m to a multiple of that length. The padding consists of the bit 1, followed by as many 0 bits as necessary, followed by a 64-bit count of the number of bits in the unpadded message. (If the 1 and count won't fit in the current block, an additional block is added.) You can use the function `padding(`*`count`*`)` in the pymd5 module to compute the padding that will be added to a *`count`*-bit message.

Even if we didn't know `m`, we could compute the hash of longer messages of the general form `m + padding(len(m)*8) +` *`suffix`* by setting the initial internal state of our MD5 function to `MD5(m)`, instead of the default initialization value, and setting the function's message length counter to the size of m plus the padding (a multiple of the block size). To find the padded message length, guess the length of m and run `bits = (`*`length_of_m`*` + len(padding(`*`length_of_m`*`*8)))*8`.

The `pymd5` module lets you specify these parameters as additional arguments to the `md5` object:

```
h = md5(state="3ecc68efa1871751ea9b0b1a5b25004d".decode("hex"), count=512)
```

Now you can use length extension to find the hash of a longer string that appends the suffix "Good advice". Simply run:

[1]For Windows machines make sure you add the path to `pydoc` to the environment variable PATH.

```
x = "Good advice"
h.update(x)
print h.hexdigest()
```

to execute the compression function over `x` and output the resulting hash. Verify that it equals the MD5 hash of `m + padding(len(m)*8) + x`. Notice that, due to the length-extension property of MD5, we didn't need to know the value of `m` to compute the hash of the longer string—all we needed to know was `m`'s length and its MD5 hash.

What to submit A `part1.txt` file where: (1) Provide the hash of the longer string that appends "Good advice" (as computed above). (2) Explain in a few sentences why the length extension attack is possible. Give a couple of real world applications that such an attack could cause a problem.

Part A.2. MD5 Collisions

MD5 was once the most widely used cryptographic hash function, but today it is considered dangerously insecure. This is because cryptanalysts have discovered efficient algorithms for finding *collisions*—pairs of messages with the same MD5 hash value.

The first known collisions were announced on August 17, 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here's one pair of colliding messages they published:

Message 1:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70
```

Message 2:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

Convert each group of hex strings into a binary file.
(On Linux or Mac OS, run `$ xxd -r -p file.hex > file`.)[2]

1. What are the MD5 hashes of the two binary files? Verify that they're the same.
   `($ openssl dgst -md5 file1 file2)`

[2]For Windows machines you can download an executable port for command `xxd` from `http://ge.tt/5jfutZq/v/0`. Then just move it to your project folder. Likewise for `openssl` command to work, you must first download OpenSSL for Windows -GNU32 version recommended.

2. What are their SHA-256 hashes? Verify that they're different.
   (`$ openssl dgst -sha256 file1 file2`)

This component is intended to introduce you to MD5 collisions; you will not submit anything for it.

2.1 Generating Collisions Yourself

In 2004, Wang's method took more than 5 hours to find a collision on a desktop PC. Since then, researchers have introduced vastly more efficient collision finding algorithms. You can compute your own MD5 collisions using a tool written by Marc Stevens that uses a more advanced technique. You can download it:

`here` (source; Linux or Mac; requires Boost)
or
`here`
(Windows executable)

Note that installing Boost –particularly for Mac OS– can be quite troublesome, therefore I suggest that, only for generating the files file1, file2 (and col1, col2 in the next section), you move to a Windows machine and run the executable. After that, resume work at a Linux or Mac OS machine.

1. Generate your own collision with this tool. How long did it take?
   (`$ time fastcoll -o file1 file2`)
2. What are your files? To get a hex dump, run `$ xxd -p file`.
3. What are their MD5 hashes? Verify that they're the same.
4. What are their SHA-256 hashes? Verify that they're different.

What to submit A text file named `generating_collisions.txt` containing your answers on the four questions.

2.2 A Hash Collision Attack

The collision attack lets us generate two messages with the same MD5 hash and any chosen (identical) prefix. Due to MD5's length-extension behavior, we can append any suffix to both messages and know that the longer messages will also collide. This lets us construct files that differ only in a binary "blob" in the middle and have the same MD5 hash, i.e. $prefix \parallel blob_A \parallel suffix$ and $prefix \parallel blob_B \parallel suffix$.

We can leverage this to create two programs that have identical MD5 hashes but wildly different behaviors. We'll use Python, but almost any language would do. Put the following three lines into a file called `prefix`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
blob = """
```

and put these three lines into a file called `suffix`:

```
"""
from hashlib import sha256
print sha256(blob).hexdigest()
```

Now use `fastcoll` to generate two files with the same MD5 hash that both begin with `prefix`. (`$fastcoll -p prefix -o col1 col2`). Then append the suffix to both (`$ cat col1 suffix > file1.py; cat col2 suffix > file2.py`). I suggest that you run this step at a Linux or Mac OS machine, since the Windows equivalent of the `cat` command (`copy`) does not always behave ideally. Verify that `file1.py` and `file2.py` have the same MD5 hash but generate different output.

Extend this technique to produce another pair of programs, `good` and `evil`, that also share the same MD5 hash. One program should execute a benign payload: `print "I come in peace."` The second should execute a pretend malicious payload: `print "Prepare to be destroyed!"`

What to submit Two Python 2.x scripts named `good.py` and `evil.py` that have the same MD5 hash, have different SHA-256 hashes, and print the specified messages. Also include a text file names `goodEvil.txt` explaining how you generate the files and why they have the same MD5 hash.

Acknowledgment

The programming part of this assignment was developed by Alex Halderman for course ECE388 at UMich.

Programming assignment - Part B: Attacking RSA

In this lab, you will investigate vulnerabilities in RSA, when it is used incorrectly.

Objectives:

- Understand how to apply RSA encryption and digital signatures.
- Investigate how using RSA incorrectly can compromise confidentiality and integrity

Part B.1. Getting Familiar with RSA

1.1 Textbook RSA

Here, we summarize a naive version of the RSA ("textbook RSA") encryption and digital signatures schemes. This version of RSA should not be used; you will show in this lab several ways in which it can be vulnerable. Consult https://en.wikipedia.org/wiki/RSA_(cryptosystem) for more detail.

Encryption Below we describe the key generation (KeyGen), encryption (Enc), and decryption (Dec) algorithms of RSA.

1. KeyGen:
   (a) Choose two random primes p and q.
   (b) Let $n = pq$ be the modulus.
   (c) Select a random encryption exponent e such that the greatest common denominator of e and $(p-1)(q-1)$ is 1 (that is, e and $(p-1)(q-1)$ are relatively prime).
   (d) Let the decryption exponent d be $e^{-1} \bmod (p-1)(q-1)$.
   (e) Return the public key $pk = (n,e)$ and the secret key $sk = d$.
2. $\mathsf{Enc}(pk = (n,e), m)$, where m is a message to be encrypted:
   (a) Return the ciphertext $c = m^e \bmod n$.
3. $\mathsf{Dec}(sk = d, c)$, where c is a ciphertext to be decrypted:
   (a) Return the recovered message $m = c^d \bmod n$.

As an example, say that during KeyGen, I choose $p = 5$ and $q = 11$. Therefore, I have $n = 55$. If I choose $e = 3$, I need to find a d such that $3d \bmod (5-1)(11-1) = 3d \bmod 40 = 1$. I can try a few numbers, and see that $3 \times 27 \bmod 40 = 81 \bmod 40 = 1$, so I set $d = 27$[3].
Now, say that you know only $n = 55$ and $e = 3$. Encryption is just raising the message m to the power of e; you can encrypt $m = 51$ as $c = 513 \bmod 55 = 132651 \bmod 55 = 46$.
Decryption is the inverse of encryption; it takes the e-th root of the ciphertext. e and d are chosen in such a way that raising to the power of d is the same as taking the e-th root. Therefore, I, who also know $d = 27$, can decrypt $c = 46$ by taking $m = 4627 \bmod 55 = 51$.
Taking the eth root of a value modulo an n whose factorization you don't know is considered to be unrealistic for large numbers. However, in our example, someone else, who doesn't know $d = 27$, can also figure out what m is, just by trying each number modulo 55 and raising it to $e = 3$. (If they get 46, they know they've found the right m!) This is only possible because we chose small p and q. If p and q are over 1000 bits long, instead of just 5 bits long, decryption without knowledge of d (or the factorization of n) becomes unrealistic.

[3] In reality, there are way more efficient ways to do this than just trying a few numbers!

Signatures In "textbook RSA" digital signatures, key generation (KeyGen) is exactly the same as it is in the encryption scheme described above. Signing (Sig) is reminiscent of decryption, and verification (Ver) is reminiscent of encryption.

1. $\mathsf{Sig}(sk = d, m)$, where m is a message to be signed:
   (a) Return the signature $\sigma = m^d \bmod n$.
2. $\mathsf{Ver}(m, \sigma, pk = (n, e))$, where m is a message and σ is a signature to be verified:
   (a) Verify that $\sigma^e \bmod n = m \bmod n$.

Using RSA Securely

In order to avoid vulnerabilities such as the ones explored in this lab, instead of "textbook RSA", RSA PKCS #1 (ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc) is used.

Encryption To encrypt a message m using RSA PKCS #1 version 1.5, the message is padded as follows:

$$\texttt{00}\quad\texttt{02}\quad\underbrace{\texttt{RR}\cdots\texttt{RR}}_{\text{at least 8 random non-zero bytes}}\quad\texttt{00}\quad\underbrace{m}_{\text{the message itself}}$$

Signatures To sign a message m using RSA PKCS #1 version 1.5, the message is padded as follows:

$$\texttt{00}\quad\texttt{01}\quad\underbrace{\texttt{FF}\cdots\texttt{FF}}_{k/8-38\text{ bytes wide}}\quad\texttt{00}\ \underbrace{\texttt{3021300906052B0E03021A05000414}}_{\text{ASN.1 ``magic'' bytes}}\quad\underbrace{\texttt{XX}\cdots\texttt{XX}}_{\text{20-byte SHA-1 digest}}$$

The result is signed using the "textbook RSA" signing function. The algorithms that check the digital signatures need to be implemented very carefully. In Part 5, you will show that if the implementation of the verification algorithm is slightly off, PKCS1.5 signatures can be forged.

NOTE: The implementation vulnerabilities that are explored in this lab have occurred often enough in the wild that many cryptographers and practitioners have concluded that using RSA PKCS v1.5 is not a good idea. RSA - OAEP is the suggested choice for RSA encryption. See https://tools.ietf.org/html/rfc3560.)
Avoiding the attacks discussed in this lab has also been cited as motivation for using elliptic curve cryptographic signatures (e.g., ECDSA) instead of RSA. Nevertheless, RSA is still widely used in practice.

1.3 Textbook RSA in Python

You can experiment with RSA yourself, using Python. In `tools.py`, we provide several functions you can use in this exploration. For instance, we provide text to integer (and integer to text) conversion, because the RSA algorithms assume that the message m is an integer modulo n.

- `text_to_int`: takes in a string and returns an integer which can be used in RSA.
- `int_to_text`: takes in an integer and returns the corresponding string.

We also provide a few other functions:

- `find_root`: takes in integers x and r, and returns the integer component of the rth root of x. (Python's built in `pow` function can be used to do this, but it doesn't work for very large 'long'-type values.)
  For instance, `find_root`$(x = 50, r = 2)$ = 7. $\sqrt{50} \sim 7.07106$, and 7 is the integer part of 7.07106.
- `combine_moduli`: takes in two integer moduli n_1 and n_2 and two integers x_1 and x_2, and returns an integer $x \bmod n_1 n_2$ such that $x \bmod n_1 = x_1$ and $x \bmod n_2 = x_2$.
  For instance, `combine_moduli`$(n_1 = 5, n_2 = 7, x_1 = 2, x_2 = 3)$ = 17, since $17 \bmod 5 = 2$, and $17 \bmod 7 = 3$.

The following shows how to use Python to do some basic RSA operations:

```
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
import tools

# set the security parameter:
secparam = 2048

# generate an RSA key:
key = RSA.generate(secparam)
public_key = key.publickey()

# save the public and private keys to files and read them back:
open('key.pub', 'w').write(public_key.exportKey())
open('key.priv', 'w').write(key.exportKey())
```

```
# read the public key in:
recovered_public_key = RSA.importKey(open('key.pub', 'r').read())
assert recovered_public_key == public_key
# read the private key in:
recovered_key = RSA.importKey(open('key.priv', 'r').read())
assert recovered_key == key

# use the PUBLIC KEY to encrypt a message:
message = "I ATE SOME PIE"
message_int = tools.text_to_int(message)
ciphertext = public_key.encrypt(message_int, None)

# save the ciphertext to a file and read it back:
open('ciphertext', 'w').write(str(ciphertext))
recovered_ciphertext = eval(open('ciphertext', 'r').read())
assert recovered_ciphertext == ciphertext

# use the PRIVATE KEY to decrypt the message:
recovered_message_int = int(key.decrypt(ciphertext))
assert recovered_message_int == message_int

# use the PRIVATE KEY to sign a hash of the message:
hash = SHA256.new(message).digest()
signature = key.sign(hash, None)

# save the signature to a file and read it back:
open('signature', 'w').write(str(signature))
recovered_signature = eval(open('signature', 'r').read())
assert recovered_signature == signature

# use the PUBLIC KEY to verify the signature:
hash = SHA256.new(message).digest()
assert public_key.verify(hash, signature)
```

We can also use math operations in Python to explore RSA ciphertexts and signatures manually. For instance, you can manually verify the signature. The modulus can be accessed as public_key.n, and the public exponent can be accessed as public_key.e.

```
print "%0128x" % pow(signature[0], public_key.e, public_key.n)
# The suffix of the signature^e mod n should be:
print SHA256.new(message).hexdigest()
```

2. Small Message Space Attack

Imagine that you are a government agent trying to determine when your neighboring country, Malland, will attack. Open `part2_ctext` to find a "textbook RSA" ciphertext sent by Malland to its ally, Horridland. You know that the Mallanders are likely to be saying one of three things:

- "attack tomorrow at dawn",
- "attack just before dusk", or
- "attack early next week".

The file `part2_key.pub` contains Horridland's public key. Use this knowledge to figure out what the message is. What would you do to prevent Malland from executing the same attack on ciphertexts you send?

What to Submit

1. A Python 2.x script named `part2.py` that: (a) Takes in three parameters: i. The path to a file with a public key, ii. The path to a file with a ciphertext, and iii. The path to a file enumerating the possible messages, each on a new line. (b) Prints the identified message. This script should be callable as follows:

   ```
   baldimtsi$ python part2.py part2_key.pub part2_ctext part2_messagespace
   The message is 'xxxxx'.
   ```

2. A file named `part2.txt` with your suggestion for how to prevent the attack you executed in `part2.py`.

3. Small Plaintext and Encryption Exponent Attack

Open `part3_ctext` to find another "textbook RSA" ciphertext, sent by Malland to Horridland. However, this time, you don't really know what Malland might be saying. You do, however, know that the encryption exponent is $e = 3$, and that Mallanders tend to be very brief and to the point, so you think the message in question is probably very short. Use this knowledge to figure out what the message is.

Defense: Padding A technique that prevents this attack is padding, which is used in PKC#1. Padding artificially increases the size of the message so that it is almost as big as the modulus n by adding extra characters or bits to the message. Why would padding help prevent this attack?

What to submit

1. A Python 2.x script named `part3.py` that: (a) Takes in one parameter: the path to a file with a ciphertext. (b) Prints the identified message. This script should be callable as follows:

   ```
   baldimtsi$ python part3.py part3_ctext
   The message is 'xxxxx'.
   ```

2. A file named `part3.txt` that explains why padding prevents the attack you executed in `part3.py`.

Submission Checklist for Programming Parts

Part A: Hash Functions

- **Part 1.1 [10 points]**: `part1.txt` with your answers.
- **Part 2.1 [8 points]**: A text file `generating_collisions.txt` with your answers to the four short questions.
- **Part 2.2 [12 points]**:Two Python 2.x scripts named `good.py` and `evil.py` and a text file `goodEvil.txt`.

Part B: RSA

- **Part 2 [15 points]**: Python script named `part2.py` and text file `part2.txt`.
- **Part 3 [15 points]**: Python script named `part3.py` and text file `part3.txt`.