UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Spring 2018**
Hassan Khan

ASSIGNMENT 3

Assignment due date: **26 July 2018 3:00 pm**

Total Marks: 50
Written Response Questions TA: Jason Cao y243cao@uwaterloo.ca
(Office hours: Tuesdays 12:00 –1:00 pm in DC 3333)
Programming Questions TA: Bailey Kacsmar bkacsmar@uwaterloo.ca
(Office hours: Friday 1:30 pm–2:30 pm in DC 3333)

Please use Piazza for all communication. Ask a private question if necessary. The TAs' office hours are also posted to Piazza for reference. You are expected to follow the expected Academic Integrity requirements for Assignments; you can find them here: https://uwaterloo.ca/library/get-assignment-and-research-help/academic-integrity/academic-integrity-tutorial. Strict penalties will be enforced on students for any Academic Integrity violations.

Written Response Questions [23 marks]

Note: Please ensure that written questions are answered using complete and grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as on the correctness of your answers.

1. [9 marks] **Cryptosystems**

   (a) [5 marks] One-Time Pad (OTP)
   OTP is defined as follows: $C = P \oplus K$, where ciphertext C, plaintext P and key K are bit strings of the same length, and $\oplus$ is bitwise XOR. OTP is not secure if the key is reused.

   i. [1 mark] Suppose two plaintext strings of the same length, P_1 and P_2, are padded with the same key K to produce ciphertexts C_1 and C_2. Describe how you can obtain $P_1 \oplus P_2$, given C_1 and C_2 (but not P_1, P_2, or K). Prove the correctness of your solution.
   $P_1 \oplus P_2 = C_1 \oplus C_2$
   Proof: $C_1 \oplus C_2 = (P_1 \oplus K) \oplus (K \oplus P_2) = P_1 \oplus (K \oplus K) \oplus P_2 = P_1 \oplus 0 \oplus P_2 = P_1 \oplus P_2$

   ii. [2 marks] Now suppose P_1 and P_2 are English ASCII texts, where every eight bits represent either a letter or whitespace. Describe how you can obtain P_1 and P_2, given $P_1 \oplus P_2$.
   Crib-dragging: use very frequent English words (1 mark), such as "the", "is", "and", put them on/drag them through every possible position in the string $P_1 \oplus P_2$, and calculate the corresponding XOR of the chosen word and the substring of $P_1 \oplus P_2$ at the position (1 mark). If one of P_1 and P_2 indeed has the word at a position, then the result calculated at this position should be part(s) of some other English word(s), which is exactly the plaintext substring of the other message at the location. Otherwise, the result is highly likely to be unprintable/unreadable. As you progress, you can also make guesses based on the partial words/sentences you already have, and try out your guesses using similar approach.

   Solutions that are computationally close to brute force: 1 mark.
   No description other than "crib-dragging": 1 mark.

   iii. [2 marks] Given C_1 and C_2, and after recovering the plaintexts using the two previous steps, can you determine the key K, and why?
   No. Either $K = P_1 \oplus C_1 (= P_2 \oplus C_2)$ (1 mark) or $K = P_1 \oplus C_2 (= P_2 \oplus C_1)$ (1 mark), but there is no way of determining which one. The reason is that for either of the plaintext messages recovered, it is not known if it corresponds to C_1 or C_2.

(b) [4 marks] (Textbook) RSA
Consider the following usage of RSA (known as "Textbook RSA"). For simplicity, the process of key generation is omitted here.

- Bob generates 3 positive integers according to the key generation algorithm: n, e, and d. He publicly announces n and e on his webpage (public key), and keeps d to himself (private key).
- When Alice wants to send message M to Bob, she calculates $C = M^e \ mod \ n$, and sends C to bob.
- When Bob receives C from Alice, he recovers M by calculating $M = C^d \ mod \ n$.

Based on this definition, answer the following questions.

i. [1 mark] Is this scheme secure against replay attacks? If so, explain why; otherwise, give a simple countermeasure.
No. (This is because given the same plaintext, the ciphertext is always the same at any time.) Countermeasure: append the current time to the plaintext before encryption.

ii. [2 marks] If the total number of possible messages is limited (for example, M is 32-bit), can an attacker recover a plaintext M given the corresponding ciphertext C and the public key (n and e)? Explain why. Also, if such an attack is possible, design a countermeasure.
Yes, dictionary attack (1 mark). Countermeasure: append a long random string to the plaintext each time before encryption (1 mark).

iii. [1 mark] What measure can we add to this encryption scheme, such that Bob can authenticate Alice when he receives C?
Message authentication code or digital signature.

2. [8 marks] **GnuPG**
A GnuPG public key for y243cao@uwaterloo.ca is provided along with the assignment on the course website (y243cao.asc). Perform the following tasks. You can install GnuPG on your own computer, or use the version we have installed on the ugster machines.

(a) [2 marks] Generate a GnuPG key pair with 2048 bits for yourself. Use RSA as the encryption algorithm, your real name, and your uwaterloo university email address. Export this key using ASCII armor into a file called **key.asc**. [Note: older versions of GnuPG might not have the RSA option, so check that the version you are using has this option. The ugster machines have a new enough version, but the student.cs machine may not.]

(b) [2 marks] Use this key to sign (not local-sign) the y243cao@uwaterloo.ca key. Its true fingerprint is: `FA4F 2ED6 347C B3A0 EE8F CDD1 82A0 9499 C648 596B`. Export your signed version of the y243cao@uwaterloo.ca key into a file called **y243cao-signed.asc**; be sure to use ASCII armor. [Note: signing a key is not the same operation as signing a message.]

(c) [2 marks] Create a message containing your userid and name. Sign it using the key you generated, and encrypt it to the y243cao@uwaterloo.ca key. You should do both the encryption and signature in a single operation. Make sure to use ASCII armor, and save the output in a file called **message.asc**.

(d) [2 marks] Briefly explain the importance of fingerprints in GnuPG. In particular, explain how users should check fingerprints and what type of attacks are possible if users do not follow this procedure properly.

Fingerprints are essential to using GnuPG, since anyone can set up a public key pair for a name/email address of their choice, regardless of their actual identity. In order to be sure you are talking to the person you believe you are talking to, you need to verify the fingerprint of a given public key. In particular, you need to get this fingerprint via a secure channel (phone, business card the person gave you in person, etc.). That way once you have the (much larger) public key, you can use the (short, trusted) fingerprint to verify that you are using the correct public key.

3. [6 marks] **Data Privacy**

(a) [3 marks] A hospital has in its database a table called `Employees`, which contains N records. This table stores the following information for each employee:

- ID: A unique alphanumerical string that is used to identify each employee in the table.
- Type: The employee's job type. All employees in this hospital are classified into three categories: Doctor, Nurse and Non-medical.
- Description: Additional description of the employee's job.
- Salary: Amount of money earned by the employee per month.

Table 1 shows some sample entries that are randomly drawn from the table.

ID	Type	Description	Salary
a45smith	Nurse	intensive care	8742
r1krold	Non-medical	accountant	6479
k5park	Nurse	emergency response	6059
j1diego	Doctor	brain surgeon	65000

Table 1: Samples drawn from the `Employee` table

To prevent users of this database from learning the salary of other employees, the database is configured to suppress the Salary field in the output of queries. However, users can execute queries in the following form:

```
SELECT SUM(Salary) FROM Employee [WHERE ...]
```

In addition, queries that match fewer than k or more than $N - k$ but not all N records are rejected.

Among all the authorized users of the hospital's database, Eve is particularly curious and wants to know the salary of a particular employee whose ID is "jdoe". She also has the following background knowledge about the hospital and the database:

- The composition of the hospital staff is 20% doctors, 60% nurses and 20% non-medical workers.
- The exact value of k is unknown, but is believed to be in the range of $(0.05N, 0.2N)$.
- Nothing else is known about "jdoe", other than this employee ID.

Given the above information, describe how Eve can use a tracker attack to query the database and derive the salary of "jdoe". In your solution, you should present 1) the tracker, 2) three additional queries, and 3) how to derive the salary of "jdoe" based on the query results. (Note: answers that do not point out the tracker will lose 1 mark.)

Tracker: `SELECT SUM(Salary) FROM Employee WHERE Type = 'Nurse';`

Queries:
`SELECT SUM(Salary) FROM Employee WHERE ID = 'jdoe' OR NOT Type = 'Nurse'`
→ gives result s_1
`SELECT SUM(Salary) FROM Employee WHERE ID = 'jdoe' OR Type = 'Nurse'`
→ gives result s_2
`SELECT SUM(Salary) FROM Employee` → gives result s_3

The salary of "jdoe" is $s_1 + s_2 - s_3$.

(b) [3 marks] The hospital is releasing some patient information in a public report. Table 2 is the full table that the hospital intends to include in the report.

Age	Gender	Disease
81	F	Flu
65	M	Flu
63	M	Heart disease
81	M	Heart disease
78	F	Heart disease
73	M	Heart disease
67	M	Heart disease
74	F	Cancer
70	M	Cancer
84	F	Cancer
77	M	Alzheimer's

Table 2: Full patient information table before anonymizing

To protect the privacy of the patients included, the hospital decides to modify the table such that it is 3-anonymous. In particular, the hospital declares that "age" and "gender" fields are identifiers, and further poses the following constraints during the anonymization process:

- The age field cannot be masked (e.g. changing "64" to "6*" is not allowed). However, it can be modified to another integer within the range of ± 5 (e.g. "64" can be changed to any integer within the range [59, 69]).
- The gender field cannot be masked or modified.

Please present a 3-anonymous solution satisfying the above constraints, and give the value l for which your solution is l-diverse. You may reorder the records in any way for your convenience.

The following table is one possible solution. It is 2-diverse.

Second group age can be 65–68, last group age can be 76–78.

Age	Gender	Disease
79	F	Cancer
79	F	Cancer
79	F	Flu
79	F	Heart disease
66	M	Cancer
66	M	Flu
66	M	Heart disease
66	M	Heart disease
77	M	Alzheimer's
77	M	Heart disease
77	M	Heart disease

Programming Questions [27 marks]

Background

In this section we will study padding oracle attacks. Block ciphers operate on a fixed number of bits, such as 64 (DES) or 128 (AES). To encrypt longer messages, a mode of operation such as CBC can be used (see Figure 1). However, this still requires that the message length be a multiple of the block size. To accommodate arbitrary-length messages, a padding scheme is used. The decryption routine will need to check that the message padding is valid. Unfortunately, information about the validity of the padding can easily be leaked to an attacker. This "padding oracle" can allow the attacker to decrypt (and sometimes encrypt) messages without knowing the encryption key.

The padding oracle attack was originally described in a 2002 paper by Serge Vaudenay. This paper is available at `https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf` and will serve as your reference for this attack. Note that the author frequently uses the term "word" where we would usually say "byte". Padding oracle attacks have continued to be relevant since their inception, with new attacks being discovered regularly. Some recent examples are the POODLE attack discovered in 2014 and the DROWN attack discovered in March 2016, both attacking SSL and TLS (not required reading).

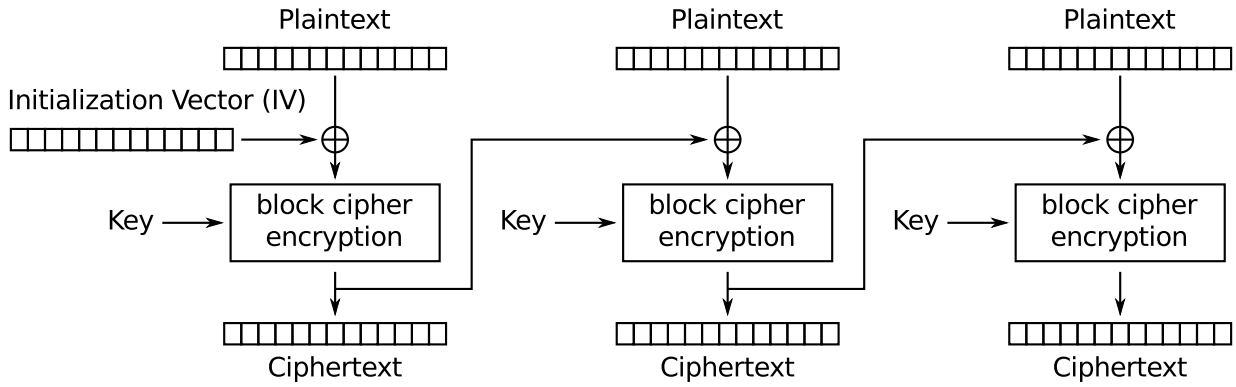

Figure 1: CBC Mode Encryption

Application Description

A simple web application is running at `http://localhost:4555` on each ugster machine. To view this application in your web browser, you can use `http://ugsterNN.student.cs.uwaterloo.ca:4555` (instructions for working from home are the same as with previous

assignments). However, please use `http://localhost:4555` in your programs to ensure that they run quickly no matter which ugster machine we test them on.

When you visit the web application it will set a cookie named `user` in your browser. In other words, the HTTP response will contain a `Set-Cookie` header like the following:

```
Set-Cookie: user="K/wLMO+V8Qbo5B4spv6/PP98W1mofu7vQT83dbascyLcyc9mlFi4qzLyYjsyTLWF"
```

The value of this cookie is encrypted with AES using a key known only to the web server. AES is used in CBC mode, and the randomly chosen IV is prepended to the ciphertext. The result is then Base64 encoded to form the cookie value. Base64 is a standard way of encoding binary data into printable ASCII characters. It is implemented in many programming languages and the `base64` Unix utility, and is specified in RFC 3548. The padding scheme used by the web application is the one suggested by NIST referred to in Section 4 of the padding oracle paper. The first byte of the padding has the value 0x01, and all other padding bytes have the value 0x00. There is always at least one byte of padding.

When you visit the web application again, your browser will send the cookie back to the server. This is done with a `Cookie` header with the same form as the `Set-Cookie` header above. This behaviour can of course be emulated programmatically. The web server will use its encryption key to attempt to decrypt any cookie sent to it. If the decryption is successful the HTTP response will contain a 200 (OK) response code. However, if the decryption fails a 500 (Internal Server Error) response code will be returned. See the section below titled "HTTP with `curl`" for more information on working with HTTP.

Questions

1. [2 marks] What are the average case and worst case number of padding oracle calls required to perform this attack?

   An N-block ciphertext can be decrypted with $NbW/2$ calls on average, where b is the block size in words and W is the number of possible words. In the standard case where a word is one byte, $W = 256$. The worst case number of calls is NbW.

2. [3 marks] What cryptographic tool could be used to fix this vulnerability? Which of the CIA properties {Confidentiality, Integrity, Authenticity} does it provide? How does it fix the vulnerability? Assume that the web server must return a different response for users with valid versus invalid cookies.

   Some form of authenticated encryption is required to ensure that data integrity is preserved. One easy way to do this is by appending a MAC to the ciphertext. The MAC is checked before the decryption stage so the attacker can't submit invalid ciphertexts.

This could also be achieved with a specialized mode such as AES-GCM. Both of these options provide all of the CIA properties.

3. [3 marks] How would you modify the attack described in the paper to account for the different padding scheme used by the web server? List the lines in Sections 3.1 and 3.2 that require changes and show how they should be changed.

   In Section 3.1, change line 5b to output $(r_{b-n+1} \oplus (n)), r_{b-n+2}, \ldots, r_b$.
   In Section 3.2, line 1 should read: take $r_k = a_k$ for $k = j, \ldots, b$.
   In Section 3.2, line 5 should read: output $r_{j-1} \oplus i \oplus 1$.
   Note that with this padding scheme you should also check the padding length for each additional byte. This is because the paddings of different lengths do not have unique suffixes.

4. [12 marks] Write a program called `decrypt` that implements the padding oracle attack on the web application described above. Your program will be called with a single command line argument containing a Base64-encoded cookie value to be decrypted. These cookies will have at least two 16-byte blocks (the IV and at least one ciphertext block), but may not have the same number as those returned by the server. Your program should generate the appropriate cookie values, send them to the web server using HTTP, and observe its response codes in order to decrypt the given cookie value. It should print the resulting plaintext (without padding) to `stdout`, which will consist of only printable ASCII characters. You can print debug output to `stderr` if you like. Your program should run in a maximum of 10 minutes for inputs up to 10 blocks long.

   You may use any programming or scripting language available on the ugster machines. If your preferred language is not available, we may be able to accommodate requests. Many programming languages have built-in libraries for making HTTP requests and encoding and decoding Base64. You can also use third party libraries for these tasks if necessary. Another option is to call the `curl` and `base64` programs as described in the section below titled "HTTP with `curl`". You will submit a single file named `src.tar`. For evaluation, this file will be extracted and we will attempt to run a script at the top level with `./decrypt <cookie>`. This script could be your program itself, or it could be a script that compiles and then runs your program. In either case it should start with an appropriate shebang line. Your submission should not contain any compiled executables. For the example cookie shown above, the invocation would be:

   ```
   ./decrypt JKAlO6GB1bOtJ1I/VzlhbJQ+MirfsnUrk9HzzunQxvQ=
   ```

5. [7 marks] Write a program called `encrypt` that encrypts arbitrary cookie values such that they will be correctly decrypted by the web application. Your program does not need to know the web application's encryption key. It will be called with a single command line argument containing a printable ASCII plaintext to be encrypted. Your program should generate the appropriate cookie values, send them to the web server using HTTP, and observe its response codes in order to encrypt the given value. It

should print the resulting Base64-encoded ciphertext to `stdout`. You can print debug output to `stderr` if you like. Your program should run in a maximum of 10 minutes for inputs up to 10 blocks long.

Submission is similar to the previous question. Your program source files should be included in the same `src.tar` file. For this question, we will attempt to run a script at the top level with `./encrypt <plaintext>`.

Hint: In CBC mode there are three stages a block goes through in decryption. It starts as a ciphertext, the ciphertext is decrypted by the block cipher to an intermediary value, and then the intermediary value is XORed with the previous ciphertext block (or the IV for the first block) to form the final plaintext value. If we know the intermediary value for a given ciphertext block, we can manipulate the IV to gain complete control over what that block will be decrypted to. If we want to produce a plaintext x, then the IV should be x XORed with the intermediary value. This idea can easily be extended to multi-block messages. Start from the last block and move backwards. Pick any value to be the ciphertext for the last block and decrypt it (using your method from the previous question) to find the corresponding intermediary value. Then calculate the IV required to produce the desired plaintext. This IV will be the ciphertext for the second-last block, and so on.

HTTP with `curl`

HTTP (Hypertext Transfer Protocol) is a simple protocol for transferring text over a network. If you've used the Internet, you've used HTTP. In this protocol, a client makes a request to a server and the server replies with a response. A request generally asks for a resource located at a particular URL, and the response provides that resource. These resources are often HTML web pages, but here we'll be using mostly plain textual content. Let's see an example of a request and a response:

```
$ curl -v http://localhost:4555
> GET / HTTP/1.1
> User-Agent: curl/7.19.7
> Host: localhost:4555
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 37
< Set-Cookie: user="8hIOODzClpAhMw/0dAalk+YrwZqmZPmuCyOmRpzyfH+fnJ0dKBtOL+NaHufa1
MQw1XE8KtKXp2ARNcIvUVFruw=="
< Server: TornadoServer/4.3
< Etag: "4b44c375286c4a668502645e4da51dd29441e4e9"
< Date: Wed, 02 Nov 2016 17:33:32 GMT
< Content-Type: text/html; charset=UTF-8
```

```
<
Hello! I'll remember when I saw you.
```

Note that some extra debug output from `curl` has been omitted. The lines starting with "`>`" are the client talking to the server, and lines starting with "`<`" show communication in the other direction. The HTTP request starts with a method; we'll only be using `GET`. It then specifies the requested path and the protocol version. The following lines contain headers which specify additional information. These are present in both requests and responses. The response starts by confirming the protocol version. It then presents the status code, which is essential for us as it is the source of our padding oracle. The main response header of interest to us is the `Set-Cookie` header. We can use the `base64` utility to decode it, although it will likely result in unprintable characters, so we'll display the binary data as a hex dump with `xxd`:

```
$ echo "8hIOODzClpAhMw/0dAalk+YrwZqmZPmuCyOmRpzyfH+fnJ0dKBtOL+NaHufa1
MQw1XE8KtKXp2ARNcIvUVFruw==" | base64 -d | xxd
00000000: f212 0e38 3cc2 9690 2133 0ff4 7406 a593  ...8<...!3..t...
00000010: e62b c19a a664 f9ae 0b23 a646 9cf2 7c7f  .+...d...#.F..|.
00000020: 9f9c 9d1d 281b 4e2f e35a 1ee7 dad4 c430  ....(.N/.Z.....0
00000030: d571 3c2a d297 a760 1135 c22f 5151 6bbb  .q<*...`.5./QQk.
```

You can see that this cookie contains four 16-byte blocks. The first is the IV, and the next three are ciphertext blocks. To send the cookie back to the server, we can do the following:

```
$ curl -v -b user=8hIOODzClpAhMw/0dAalk+YrwZqmZPmuCyOmRpzyfH+fnJ0dKBtOL+NaHufa1
MQw1XE8KtKXp2ARNcIvUVFruw== http://localhost:4555
> GET / HTTP/1.1
> User-Agent: curl/7.47.0
> Host: localhost:4555
> Accept: */*
> Cookie: user=8hIOODzClpAhMw/0dAalk+YrwZqmZPmuCyOmRpzyfH+fnJ0dKBtOL+NaHufa1
MQw1XE8KtKXp2ARNcIvUVFruw==
>
< HTTP/1.1 200 OK
< Date: Wed, 02 Nov 2016 17:38:13 GMT
< Content-Length: 47
< Etag: "86190721ce68176612428946d30ccc0d6e5a99ec"
< Content-Type: text/html; charset=UTF-8
< Server: TornadoServer/4.3
<
Hello! I first saw you at: 2016-11-02 13:33:32
```

It would be best to use an HTTP library for your programming language when writing your solutions, but you can call the `curl` program directly as a last resort. `curl` is also very useful for manually debugging web applications.

What to hand in

Using the "submit" facility on the student.cs machines (**not** the ugster machines or the UML virtual environment), hand in the following files:

a3.pdf: A PDF file containing your answers to the written response and relevant programming questions. **a3.pdf** must contain, at the top of the first page, your name, UW userid, and student number.

src.tar: A tar archive directly containing your `decrypt`, `encrypt` program source files. (Please notice that these programs need to be executables, and without any extensions, e.g., `decrypt.sh` **is wrong** and should be sent as `decrypt`. To create the tarball, `cd` to the directory containing your code and run the command

```
tar cvf src.tar .
```

(including the `.`) in the directory where you have your files, **not** the parent directory.

key.asc, y243cao-signed.asc, message.asc: The three files required for Question 2 (GnuPG) in the written part.