UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Spring 2023**
Diogo Barradas
Adithya Vadapalli

ASSIGNMENT 3

Assignment release: **Thursday, July 6, 2023**
Milestone due date: **Wednesday, July 19, 2023 at *3:00 pm***
Assignment due date: **Tuesday, August 1, 2023 at *3:00 pm***

Total Marks: 92 (44 written + 48 programming)
Bonus Marks: 0 (no bonus marks)

Written Response TA: Sara Sarfaraz
Programming TA: Hossam ElAtali

TA Office hours (written): Thursdays 12:00 – 1:00 pm, online on BBB
TA Office hours (programming): Wednesdays 12:00 – 1:00 pm

Please use Piazza for all communication. Ask a private question if necessary.

Note that the Milestone is MANDATORY and includes Q1 and Q2 of the programming questions

What to hand in

For **programming questions**, the assignment website automatically updates your unofficial marks and records the completion timestamps whenever you successfully address a task. You can see your unofficial mark breakdown in the dashboard of the assignment website. The grade becomes official once your code submission has been examined. Please make sure you complete the tasks on the website and *submit your code* before the deadlines.

For **written questions** and **code for programming questions**, the assignment submission takes place on the `linux.student.cs` machines (**not** ugster or the virtual environments), using the submit utility. Log in to the linux student environment, go to the directory that contains your solution, and submit using the following command:

`submit cs458 3 .` (dot at the end)

CS 658 students should also use this command and ignore the warning message.

By the **milestone deadline**, you should hand in:

- **a3-milestone.tar**: Your source files for the **Questions 1 and 2 of the programming questions**, in your supported language of choice, inside a tarball (created as outlined below). **Note that this milestone is mandatory!!** You will not be able to submit the tarball for these questions after the milestone deadline.

By the **assignment deadline**, you are required to hand in:

- **a3-written.pdf**: A PDF file containing your answers for the written-response questions.
- **a3-code.tar**: Your source files for the programming assignment, in your supported language of choice, inside a tarball. While we will not run your code, it should be clear to see how your code can issue web requests to the API to solve each question. If it is not obvious to see how you solved a question, then you will not receive the marks for that question, even if marks were awarded on the assignment website.

  To create the tarball, `cd` to the directory containing your code, and run the command

  ```
  tar cvf filename.tar .
  ```

  (pay attention to the `.` at the end).

Programming Questions [48 marks]

Note: This assignment is long because it includes a lot of text to *help* you through it.

The use of strong encryption in personal communications may itself be a red flag. Still, the U.S. must recognize that encryption is bringing the golden age of technology-driven surveillance to a close.

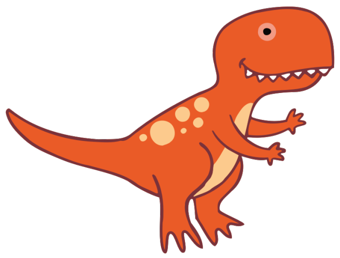
MIKE POMPEO
CIA director

Encryption works. Properly implemented strong crypto systems are one of the few things that you can rely on.

EDWARD SNOWDEN

In this assignment, you will use "strong encryption" to send secure messages. Each question specifies a protocol for sending secure messages over the network. For each question, you will use the `libsodium`[1] cryptography library in a language of your choice to send a message through a web API.

For the assignment questions, you will send messages to and receive messages from a fake user named *Rex* in order to confirm that your code is correct.



Rex

Assignment website: `https://hash-browns.cs.uwaterloo.ca`

The assignment website shows you your unofficial grade for the programming part; the grade becomes official once your final code submission has been examined. Your unofficial grade will update as you complete each question, so you will effectively know your grade *before* the deadline.

[1] `https://libsodium.org/`

The assignment website also allows you to debug interactions between your code and the web API.

`libsodium` documentation

The official documentation for the `libsodium` C library is available at this website:

`https://doc.libsodium.org/`

You should primarily use the documentation available for the `libsodium` binding in your language of choice. However, even if you are not using C, it is occasionally useful to refer to the C documentation to get a better understanding of the high-level concepts, or when the documentation for your specific language is incomplete.

Choosing a programming language

Since we will not be executing your code (although we will read it to verify your solution), you may theoretically choose any language that works on your computer.

You will need to use `libsodium` to complete the assignment. While `libsodium` is available for dozens of programming languages (check the list of language bindings[2] to find an interface for your language), you will need to limit your language choice as not all `libsodium` language bindings support all of the features needed for this assignment. Specifically, you should quickly check the `libsodium` documentation for your language to ensure that it gives you access to the following features:

- "Secret box": secret-key authenticated encryption using XSalsa20 and Poly1305 MAC
- "Box": public-key authenticated encryption using X25519, XSalsa20, and Poly1305 MAC
- "Signing": public-key digital signatures using Ed25519
- "Generic hashing": using BLAKE2b

You should also choose a language that makes the following tasks easy:

- Encoding and decoding `base64` strings
- Encoding and decoding hexadecimal strings
- Encoding and decoding JSON data

[2]`https://doc.libsodium.org/bindings_for_other_languages/`

- Sending `POST` requests to websites using HTTPS

While you are not required to use a single language for all solutions, it is best to avoid the need to switch languages in the middle of the assignment.

You may use generic third-party libraries, but of course you may not copy code from other people's solutions to this (or similar) assignment. If you use code from somewhere else, be sure to include prominent attribution with clear demarcations to avoid plagiarism.

We have specific advice for the following languages, which we have used for sample solutions:

- **Python**: This language works very well. Use the `nacl` module (`https://readthedocs.org/projects/lmctvpynacl/downloads/pdf/use-autodoc-for-apis/`) to wrap `libsodium`. The box and secret box implementations include nonces in the ciphertexts, so you do not need to manually concatenate them. The `base64`, `json`, and `requests` modules from the standard library work well for interacting with the web API.
- **PHP**: This language works well if you are already familiar with it. Use the `libsodium` extension (`https://github.com/jedisct1/libsodium-php`) for cryptography. The `libsodium-php` extension is included in PHP 7.2. Otherwise, you may need to install the `php-dev` package and install the `libsodium-php` extension through `PECL`. In that case, you must manually include the compiled `sodium.so` extension in your CLI `php.ini`. Interacting with the web API is easy using global functions included in the standard library: `pack` and `unpack` for hexadecimal conversions, `base64_encode` and `base64_decode`, `json_encode` and `json_decode`, and either the `curl` module or HTTP context options for submitting HTTPS requests.
- **Java**: This language is a reasonable choice if you are comfortable using it. The `libsodium-jna` binding (`https://github.com/muquit/libsodium-jna`) contains all of the functions that you will need. Please follow the installation instructions listed in the website. The `java.net.HttpURLConnection` class works for submitting web requests. Base64 and hexadecimal encoding functions are available in `java.util`, and JSON encoding functions are available in `org.json`. Note that the ugsters may not contain updated packages for Java.
- **C**: While C has the best `libsodium` documentation, all of the other tasks are more difficult than other languages. The assignment is also much more challenging if you use good C programming practices like error handling and cleaning up memory. If you choose C, you will spend a significant amount of time solving Question 1 before receiving any marks. We recommend `libcurl` (`https://curl.se/libcurl/`) for submitting API requests, `Jansson` (`https://github.com/akheron/jansson`) for processing JSON, and `libb64` (`http://libb64.sourceforge.net/`) for `base64` handling. Note that you will need to search for the proper usage of the `cencode.h` and `cdecode.h` headers for `base64` processing. You will need to provide your own code for hexadecimal conversions; it is acceptable to copy code from the web for this purpose, but be sure to attribute

its author using a code comment.

You may use any other language, but then we cannot provide informed advice for language-specific problems. We also cannot guarantee that bindings for other languages contain all required features.

Ugster availability

Some of the aforementioned programming languages (C, Python, PHP) and libraries for libsodium will be made available on the Ugsters in case you do not have access to a personal development computer. Note that we do not have updated packages for Java on the ugsters.

Question 1: Using the API [10 marks]

In this first question, we will completely ignore cryptography and instead focus on getting your code to communicate with the server.

Begin by visiting the assignment website and logging in with your WatIAM credentials. You will be presented with an overview of your progress for the assignment. While you can simply use a web browser to view the assignment website, your code will need to communicate with the *web API*. The web API does not use WatIAM for authentication. Instead, you will need an "API token" so that your code is associated with you.

Click on the "Show API Token" button on the assignment website to retrieve your API token. **Do not share your API token with anyone else**; if you suspect that someone else has access to your token, use the "Change API Token" button to generate a new one, and then inform the TA. Your code will need to use this API token to send and receive messages.

Question 1 part 1: send a message [6 marks]

Your first task is to send an unencrypted message to Rex using the web API. To do this, submit a web request with the following information:

- URL: `https://hash-browns.cs.uwaterloo.ca/api/plain/send`
- HTTP request type: `POST`
- `Accept` header: `application/json`
- `Content-Type` header: `application/json`

For every question in this assignment, the request body should be a JSON object. The JSON object must always contain an `api_token` key with your API token in hexadecimal format.

To send a message to Rex, your JSON object should also contain `recipient` and `msg` keys. The `recipient` key specifies the username for the recipient of your message; this should be set to `Rex`. The `msg` key specifies the message to send, encoded using `base64`.

You will receive marks for sending any non-empty message to Rex (sadly, Rex is a script that lacks the ability to understand the messages that you send). For example, to send a message containing "Hello, World!" to Rex, your request would contain a request body similar to this:

```
{"api_token": "3158a1a33bbc...9bc9f76f",
"recipient": "Rex", "msg": "SGVsb...kIQ=="}
```

Consult the documentation for your programming language of choice to determine how to construct these requests.

The web API always returns JSON data in its response. If your request completed successfully, the response will have an HTTP status code of `200` and you will receive an empty object; check the assignment website to verify that you have been granted marks for completing the question. If an error occurs, the response will have an HTTP status code that is **not** `200`, and the JSON response will contain an `error` key in the object that describes the error.

If you are having difficulty determining why a request is failing, you can enable debugging on the assignment website. When debugging is enabled, all requests that you submit to the web API will be displayed on the assignment website, along with the details of any errors that occur. If debugging is enabled and you are not seeing requests in the debug log after running your code, then your code is not connecting to the web API correctly.

Question 1 part 2: receive a message [4 marks]

Next, you will use the web API to receive a message that Rex has sent to you. To do this, submit a `POST` request to the following URL:

```
https://hash-browns.cs.uwaterloo.ca/api/plain/inbox
```

All requests to the web API are `POST` requests with the `Accept` and `Content-Type` headers set to `application/json`; only the URL and the request body changes between questions. The JSON object in the request body for your `inbox` request should contain only your `api_token`.

The response to your request is a JSON-encoded array with all of the messages that have been sent to you. Each array element is an object with `msg_id`, `sender`, and `msg` keys. The `msg_id` is a unique number that identifies the message. The `sender` value is the username that sent the message to you. The `msg` value contains the `base64`-encoded message.

Decode the message that Rex sent you. The message should contain recognizable English words. **The messages from Rex are meaningless and randomly generated.** We use English words so that it is obvious when your code is correct, but the words themselves are completely random.

To receive the marks for this part, go to the assignment website and open the "Question Data" page. This page contains question-specific values for the assignment and allows you to submit answers to certain questions. Enter the decoded message that Rex sent to you in the "Question 1" section to receive your mark.

Question 2: Pre-shared Key Encryption [10 marks]

In this part, you will extend your code from question 1 to encrypt messages using secret-key encryption. For now, we assume that you and Rex have somehow securely shared a secret key at some point. You will now exchange messages using that secret key.

Begin by importing an appropriate language binding for `libsodium`. Since every language uses slightly different notations for the `libsodium` functionality, you will need to consult the documentation for your language to find the appropriate functions to call.

Question 2 part 1: send a message [6 marks]

Send a request to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/psk/send
```

Here, `psk` stands for "pre-shared key". The format of this `send` request is the same as in question 1 part 1, except that the `msg` value that you include in the request body JSON will now be a ciphertext that is then `base64` encoded.

To encrypt your message, you should use the "secret box" functionality of `libsodium` to perform secret-key authenticated encryption. This type of encryption uses the secret key to encrypt the message using a stream cipher (XSalsa20) and to attach a message authentication code (Poly1305) for integrity. `libsodium` makes this process transparent; simply calling `crypto_secretbox_easy` (or the equivalent in non-C languages) will produce both the ciphertext and the MAC, which the library refers to as "combined mode".

You will need to generate a "nonce" ("number used once") in order to encrypt the message. The nonce should contain randomly generated bytes of the appropriate length. The size of the nonce is constant and included in most libsodium bindings (the `libsodium` documentation contains examples). To generate the message, you should `base64` encode a concatenation of the nonce followed by the output of `crypto_secretbox_easy`. Some language bindings will automatically do this for you, so check to see if the output of the function contains the nonce that you passed into it.

Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "recipient": "Rex", "msg":
  base64encode(concat(nonce, secretbox(plaintext, nonce, key)))}
```

To receive marks for this part, send an encrypted message to `Rex` using the secret key found in the "Question 2" section of the "Question Data" page. Note that the secret key is given in hexadecimal notation; you will need to decode it into a binary string of the appropriate length before passing it to the `libsodium` library.

Question 2 part 2: receive a message [4 marks]

Rex has sent an encrypted message to you using the same format and key. Check your inbox by requesting the following web API page in the usual manner:

```
https://hash-browns.cs.uwaterloo.ca/api/psk/inbox
```

You will need to decrypt this message by decoding the `base64` data, extracting the nonce (unless your language binding does this for you), and calling the equivalent of `crypto_secretbox_easy_open`. Enter the decrypted message in the "Question 2" section of the "Question Data" page to receive your marks. The decrypted message contains recognizable English words.

Question 3: Key Derivation [8 marks]

Using the same key in multiple contexts can lead to everything being encrypted, with that key being exposed if an adversary gets a hold of it. To reduce this risk, different contexts can be isolated by using different *subkeys*, which are derived from a single master key. If an adversary obtains a subkey, they cannot use it to compute the master key.

In this question, we will derive two subkeys from a single master key, and then use the subkeys to send and receive two different messages.

First, you will need to derive the two subkeys from the master key. Visit the “Question Data” page and retrieve the master key and “contexts” from the “Question 3” section. Here, the contexts are simply 8-character strings, usually identifying what each subkey is used for. Use the `crypto_kdf_derive_from_key` function in `libsodium` to obtain two *32-byte* subkeys with IDs 1 and 2 corresponding to the two contexts.

The Python wrapper for `libsodium` does not have an equivalent `crypto_kdf_derive_from_key` function, but the `blake2b` function can be used to perform the same functionality. The subkey ID is used as the salt, and the context is used as the personalization. Refer to `https://pynacl.readthedocs.io/en/latest/hashing/#key-derivation` for details.

Question 3 part 1: send a message [4 marks]

Send a request to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/kd/send
```

The format of your request should be the same as in question 2 part 1, but using the first of the subkeys you just derived.

Question 3 part 2: receive a message [4 marks]

Rex has sent an encrypted message to you using subkey 2. Check your inbox by requesting the following web API page in the usual manner:

```
https://hash-browns.cs.uwaterloo.ca/api/kd/inbox
```

As in question 2 part 2, decrypt the message by decoding the `base64` data, extracting the nonce, and calling the equivalent of `crypto_secretbox_easy_open`. Enter the decrypted message in the “Question 3” section of the “Question Data” page to receive your marks.

Question 4: Digital Signatures [10 marks]

In most common conversations, the communication partners do not have a pre-shared secret key. For this reason, public-key cryptography (also known as asymmetric cryptography) is very useful. The remaining questions focus on public-key cryptography.

In this question, you will send an unencrypted but digitally signed message to Rex.

Question 4 part 1: upload a public verification key [4 marks]

The first step in public-key communications is *key distribution*. Everyone must generate a secret *signature key* and an associated public *verification key*. These verification keys must then be distributed somehow. For this assignment, the web API will act as a *public verification key directory*: everyone can upload a verification key and request the verification keys that have been uploaded by other users.

`libsodium` implements public-key cryptography for digital signatures as part of its `sign` functions. Before sending a message to Rex, you will need to generate a signature and verification key (together called a *key pair*). Generate this pair using the equivalent of the C function `crypto_sign_keypair` in your language. You should save the secret signature key somewhere (e.g., a file), because you will need it for the next part. To receive marks for this part, upload the verification key to the server by sending a `POST` request with the usual headers to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/signed/set-key
```

The request body should contain a JSON object with a `public_key` value containing the `base64` encoding of the verification key. For example, your request body might look like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f",
  "pubkey": "CazwYZnnnYqMI6...wTWk="}
```

Upon success, the server will return a `200` HTTP status code with an empty JSON object in the body. If you submit another `set-key` request, it will overwrite your existing verification key.

Question 4 part 2: send a message [6 marks]

Now that you have uploaded a verification key, others can use it to verify that signed messages really were authenticated by you (or someone else with your secret key). Send an unencrypted and signed message to Rex by sending a request to the following web API page in the usual way:

```
https://hash-browns.cs.uwaterloo.ca/api/signed/send
```

The `msg` value in your request body should contain the `base64` encoding of the plaintext and signature in "combined mode". In the C library, you can generate the "combined mode" signature using the `crypto_sign` function. Rex will be able to verify the authenticity of your message using your previously uploaded verification key.

Question 5: Public-Key Authenticated Encryption [10 marks]

While authentication is an important security feature, the approach in question 4 does not provide confidentiality. Ideally, we would like both properties. `libsodium` supports authenticated public-key encryption, which allows you to encrypt a message using the recipient's public key and authenticate the message using your secret key. Note that for authenticated encryption, the public key is used for both encryption and for verification, while the secret key is used for both decryption and for signing.

The `libsodium` library refers to an authenticated public-key ciphertext as a "box" (in contrast to the "secret box" used in questions 2). Internally, `libsodium` performs a *key exchange* between your secret key and Rex's public key to derive a shared secret key. This key is then used internally to encrypt the message with a stream cipher and authenticate it using a message authentication code.

Question 5 part 1: verify a public key [4 marks]

One of the weaknesses of public key directories like the one implemented by the web API in this assignment is that the server can lie. If Rex uploads a public key and then you request it from the server, the server could send you *its* public key instead. If you then sent a message encrypted with that key, then the server would be able to decrypt it; it could even re-encrypt it under Rex's actual public key and then forward it along (acting as a "man in the middle").

To defend against these attacks, "end-to-end authentication" requires verifying that you received Rex's public key. This is a challenging problem to solve in a usable way and is the subject of current academic research. One of the most basic approaches is to exchange a "fingerprint" of the real public keys through some other channel that an adversary is unlikely to control (e.g., on business cards or through social media accounts).

For this part, you must download Rex's public key from the web API and then verify that you were given the correct one. Submit a `POST` request in the usual way to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/pke/get-key
```

Here, `pke` means "public-key encryption". Your request body should contain a JSON object with a `user` key containing the username associated with the public key you're requesting (in this case, `Rex`). The server's response will be a JSON object containing `user` (the requested username) and `pubkey`, a `base64` encoding of the user's public encryption key.

To verify that you received the correct public key, you should derive a "fingerprint" by passing the key through a cryptographic hash function. Use the BLAKE2b hash function provided by `libsodium` for this purpose. The C library implements this as `crypto_generichash`, but other languages might name it differently. Do not use a key for this hash (it needs to be unkeyed so that everyone gets the same fingerprint). Remember to `base64` decode the public key before hashing it! The resulting hash is what you would compare to the one that Rex securely gave to you. To get the marks for this part, enter the hash of the public encryption key, in hexadecimal encoding, into the "Question 5" section of the "Question Data" page. Keep in mind this hash/fingerprint is not the same as the public key itself.

Question 5 part 2: send a message [4 marks]

Before sending a message to Rex, you will first need to generate and upload a public key. While the key pairs generated in question 4 were generated with the `sign` functions of `libsodium`, the key pairs for this question must be generated with the `box` functions. This difference is because the public encryption keys for this question will be used for authenticated encryption rather than digital signatures, and so different cryptography is involved.

Generate a public and secret key using the equivalent of the C function `crypto_box_keypair` in your language. Then, using the same request structure as in question 4 part 1, upload your public key to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/pke/set-key
```

Once you have successfully uploaded a key (indicated by a `200` HTTP status code and an empty JSON response), you can send a message to Rex. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. This function takes as input Rex's public key (which you downloaded in the previous part, the value retrieved from a request to api/pke/get-key after it is base64-decoded), your secret key, and a nonce. The function outputs the combination of a ciphertext and a message authentication code.

You should generate the nonce randomly and prepend it to the start of the ciphertext in the same way that you did for question 2 part 1. Encode the resulting data with `base64` encoding. Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "recipient": "Rex",
                    "msg": base64encode(
concat(nonce, box(plaintext, nonce, Rex_public, your_secret))
                             )}
```

Finally, send the message to Rex in the usual way using the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/pke/send
```

Question 5 part 3: receive a message [2 marks]

To receive the mark for this part, you will need to decrypt a message that Rex has sent to you. Check your inbox in the usual way with a request to the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/pke/inbox
```

To decrypt the message from Rex, you will need your secret key and Rex's public key. After `base64` decoding the message, decrypt it using the equivalent of the C function `crypto_box_open_easy` in your language. Provide the decrypted message in the "Question 5" section of the "Question Data" page.

Written Response Questions [44 marks]

Q1: Diffie-Hellman [10 marks]

In the lectures, you saw the Diffie-Hellman key-exchange protocol. The protocol works with the integers modulo a large prime p and uses a generator $g \in \{1, ..., p\}$. Now, we will define three variants of the protocol each including the following steps:

- Alice samples $a \leftarrow \{1, ..., p\}$ at random and sends Bob $A = g^a \mod p$.
- Bob samples $b \leftarrow \{1, ..., p\}$ at random and sends Alice $B = g^b \mod p$.

1. [2 marks] For the first variant, suppose that Alice and Bob plan to use $g^{a+b} \mod p$ as their shared secret. Is this variant correct and secure? If your answer is no, describe what is wrong with this plan.
2. [2 marks] As the second variant, suppose that Alice and Bob plan to use $g^{(a^b)} \mod p$ as their shared secret. Is this variant correct and secure? If your answer is no, describe what is wrong with this plan.
3. [4 marks] As the last variant, suppose that Alice and Bob plan to use $(g^{ab} + g^a + g^b + 127) \mod p$ as their shared secret. Is this variant correct and secure? If your answer is no, describe what is wrong with this plan.
4. [2 marks] What happens if Mallory comes along and behaves as an active Man-In-The-Middle attacker? Can she manipulate the Diffie-Hellman protocol to obtain all of the plaintext communications between Alice and Bob? How?

Q2: Signatures [4 marks]

In the lectures, you saw the RSA encryption scheme. In an RSA signature scheme, the signature on message m is computed by $s = (h(m))^d \mod n$, where d is the private key of the signer and h is a hash function.

1. [1 mark] Consider a revised scheme where the signature is computed as $s' = m^d \mod n$ instead. Show that it is possible to forge signatures for some messages in this scheme.
2. [1 mark] Suppose Alice wants to sign a contract with Mallory using an RSA signature and suppose that the hash function h is a perfectly random hash function, meaning that h is selected uniformly at random from all functions mapping $\{0,1\}^* \rightarrow \{0,1\}^k$. Mallory has found 30 distinct places where she can make a slight change in the contract such that Alice does not notice: adding a space at the end of a line, adding a comma, or replacing it with equivalent words. How many possible contract versions can Mallory generate such that Alice would not notice the difference and would be willing to sign?
3. [1 mark] Mallory creates another contract reflecting an increase in the amount that Alice has to pay to Mallory. She computes the hash of the fake contract and finds a slightly different version of the original contract that hashes to the same value (We know that Alice will sign this version because she will not notice the difference). What is the probability that Mallory can find such a collision if the output size of the hash function is k bits?
4. [1 mark] What can Mallory do by finding such a collision to force Alice to pay an increased amount?

Q3: TLS and HTTPS [14 marks]

HTTPS uses the Transport Layer Security (TLS) protocol to establish a secure channel between the client and the server. TLS encrypts communication between the two parties.

1. [2 marks] The client and the server first conduct a handshake in the TLS protocol. What cryptographic information do the server and the client agree on during the handshake?
2. [2 marks] The TLS protocol uses both symmetric and asymmetric encryption. Briefly describe what each type of encryption is used for in the TLS protocol and why it is preferred for that purpose over the other type.
3. [2 mark] Briefly describe how Certification Authorities (CAs) support clients in authenticating servers.
4. [4 marks] Suppose that Bob is a terrible webmaster and chooses an invalid TLS certificate for his online store website. Alice trusts her friend Bob and chooses to ignore her browser's warning and proceeds to Bob's website to buy some stuff. An attacker, Mallory, can perform a man-in-the-middle attack in this scenario to steal Alice's credentials and banking information.

You know that man-in-the-middle attacks involve the attacker establishing a channel with both ends — the server (Bob) and the client (Alice). Note that Mallory might need to perform some preparation even before Alice's attempt to connect to Bob's website.

(a) [2 marks] Specify what cryptographic information Mallory needs to generate or modify so that Alice's vulnerable browser (the client) will establish a secure channel with Mallory.

(b) [2 marks] Mallory needs to convince Alice that she is communicating with Bob's server to reassure her that her purchase was successful. Present a *clear and concise* protocol that Mallory executes in order to convince Alice of a successful purchase and to steal her banking credentials. Mallory will need to communicate with the server and relay (repeat) information back to Alice and vice versa. So, in the steps of your protocol, include what Mallory encrypts or decrypts, under a key shared with one of Alice or Bob and sends to/received from them.

5. [4 marks] Bob has a long history of ordering trading cards from supercards.com. Typically, Bob first connects to the HTTP version of this website before getting automatically rerouted by the web server itself to the HTTPS version of the website. How can an attacker leverage this fact to obtain Bob's credit card details when Bob issues an order? Name one way how Bob could prevent such an attack and one way how the attack could be prevented by the supercards.com webserver.

Q4: Database Privacy [5 marks]

k-Anonymity

Alice and Co. is a cutting-edge company dedicated to advancing privacy practices worldwide. To showcase its success, the company publishes information about racial diversity among its employees, claiming to protect their privacy by modifying the initial data set into a k-anonymous one. Table 1 is the table they wish to anonymize and publish:

The Name, Age, and Gender columns are considered identifiers. In order to protect the privacy of the employees, the company's anonymization team has placed the following conditions on the data and decided to modify the table such that it is 3-anonymous:

- Name is fully masked;
- Race cannot be masked;
- Age must be generalized to at least 3 **equally sized** ranges.

Based on these constraints, the anonymization team has chosen the following ranges for age: [21 - 25], [26 - 30], [31 - 35]. Table 2 shows the anonymized table.

1. [3 marks] Is the anonymized table 3-anonymous? If so, explain why and give the value for ℓ for

Name	Age	Gender	Race
Lin	34	F	Asian
Jun	33	F	Asian
John	21	M	White
Jennifer	35	F	Black
Jack	26	M	American Indian
Bob	22	M	American Indian
Lily	27	F	Asian
Lisa	29	F	Black
Kevin	23	M	Black
Sarah	27	F	White

Table 1: Employees' racial diversity

Name	Age	Gender	Race
*	[31 - 35]	F	Asian
*	[31 - 35]	F	Asian
*	[21 - 25]	M	White
*	[31 - 35]	F	Black
*	[26 - 30]	M	American Indian
*	[21 - 25]	M	American Indian
*	[26 - 30]	F	Asian
*	[26 - 30]	F	Black
*	[21 - 25]	M	Black
*	[26 - 30]	F	White

Table 2: Employees' racial diversity (Anonymized)

which the table is ℓ-diverse. If not, present a solution where the table is exactly 3-anonymous (i.e., neither more nor less than 3-anonymous) by *only* changing the value ranges used for the Age identifier. Then, provide the value of ℓ for which the new table is ℓ-diverse. Recall that you should respect the same constraints used to create Table 2, namely:

- Name is fully masked;
- Gender cannot be masked;
- Age must be generalized to at least 3 **equally sized** ranges.

After a conflict between Alice and her partners, a new anonymization team was hired to replace the previous one. This fresh team had a different approach: They aimed to make the published Table 4-anonymous, providing even stronger privacy protection. The new anonymization team chose the following ranges for age:[18 - 26], [27 - 35]. Table 3 shows the 4-anonymized table. It's important to note that this table does not represent the same set of employees as before due to changes in the workforce.

2. [1 marks] Suppose Tom discovered on LinkedIn that John has not been fired and therefore is

Name	Age	Gender	Race
*	[27 - 35]	F	Middle Eastern
*	[27 - 35]	F	White
*	[18 - 26]	M	White
*	[27 - 35]	F	Black
*	[18 - 26]	M	Middle Eastern
*	[18 - 26]	M	Asian
*	[27 - 35]	F	Asian
*	[27 - 35]	F	Black
*	[18 - 26]	M	Asian
*	[27 - 35]	F	Asian

Table 3: Employees' racial diversity, 4-Anonymized

in both tables. What can Tom learn from John Based on the two anonymized tables that he did not know before?

3. [1 marks] It seems background knowledge can be a real problem for Alice and Co. and its employees. Can you suggest a popular database privacy notion that is resilient to background knowledge?

Q5: Private Information Retrieval [11 marks]

Alice is a privacy-conscious individual who is fed up of targeted advertising and hyper-specific recommendations. PrivateMart Co. is a *seemingly* privacy-friendly shopping company who wishes to cater to individuals like Alice.

iCompany's inventory of items for sale is presented in the form of an $n \times m$ matrix A. Row i corresponds to the product with ID i. Each row of the matrix contains m elements; each element specifies the product somehow (for example, its price and weight).

The original (non-private) scheme for a user to retrieve information about a product, given by row c of the matrix A, was as follows: the user sends the row number c to PrivateMart. PrivateMart returns row c of the matrix A to the user. Maliciously, PrivateMart would then log the number c that each user sent and build up a detailed profile of their search history.

Note that sending a row number c to PrivateMart is mathematically equivalent to left-multiplying matrix A by a unit row vector of length n with a 1 in index c (that is, a vector with 0s as its elements in all indices other than index c). This vector is known as the query vector. So, from here on, we describe the original scheme as follows: the user sends a query vector $\vec{q}$ and PrivateMart computes $\vec{q} \cdot A$. That is, PrivateMart multiplies the query vector with the inventory matrix. PrivateMart then sends the resulting row to the user.

So, for $c = 2$, and a 3×2 matrix A, we get:

$$\vec{q} \cdot A = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} = \begin{bmatrix} A_{21} & A_{22} \end{bmatrix}$$

Note : Addition operations as part of matrix multiplication are performed bitwise modulo 2; i.e., the additions are effectively XOR operations.

Privacy-conscious individuals like Alice have called for a revolution: companies like PrivateMart should be prevented from learning what the user queries, at *all* costs.

1. [2 marks] Imagine the most *trivial* solution that would fulfill the revolution's goals. In other words, Alice should obtain the product $\vec{q} \cdot A$ such that PrivateMart does not learn $\vec{q}$. This solution may involve Alice *downloading* a lot of matrix elements, and it should not involve any cryptography. This solution does not need to be based on the original query scheme above. Briefly describe the solution. How many matrix elements does Alice need to *upload* in this solution?
2. [2 marks] PrivateMart Co. is concerned about its bandwidth usage, so they want to count the total number of matrix elements that are downloaded by its customers. Given the two following settings:
   (a) Alice wants to obtain the details of $n/2$ products.
   (b) $n/2$ customers at different locations want to obtain the details of one product each.
   Calculate the total number of matrix elements downloaded using your trivial scheme in each setting. In which setting does the trivial scheme involve downloaded a *smaller* number of matrix elements? Explain your reasons why.

PrivateMart Co. has to spend a lot of money on internet bills to sustain the above trivial scheme. It wants to ensure that Alice can privately retrieve content without incurring high communication costs. In Module 5, part 11 (PETs), we have introduced Private Information Retrieval (PIR). In this question, we'll be exploring 3-server information-theoretic PIR (IT-PIR).

So, PrivateMart has negotiated with two other companies; each company will host a server that contains an identical copy of A. (So now, we have a total of three servers.) You know that IT-PIR requires assuming that each of these servers does not share the requests that they received or the responses that they will send back, and so you may make this non-collusion assumption. Using these 3 servers, PrivateMart Co. proposes the following solution for Alice to retrieve row c:

(a) As usual, the user generates a query vector $\vec{q}$ that is 1 at element c and 0 otherwise. That is,

$$\vec{q} = \begin{cases} 1 & \text{if } i = c, \\ 0 & \text{otherwise} \end{cases}$$

But the user doesn't directly send $\vec{q}$ to the 3 servers.

(b) The user generates two uniformly random bit vectors $\vec{q_1}$ and $\vec{q_2}$, of size n. (Each bit is flipped at random for each element of each vector.)

(c) The user generates a vector $\vec{q_3}$ by XOR'ing each element from $\vec{q_1}$ and $\vec{q_2}$. That is, $\vec{q_3}$ is obtained using the following formula where $\oplus$ represents the XOR operator.

$$\vec{q_3} = \vec{q_1} \oplus \vec{q_2} \oplus \vec{q}$$

(d) The user uploads $\vec{q_1}$ to new server 1, $\vec{q_2}$ to new server 2 and $\vec{q_3}$ to PrivateMart's original server.

(e) Server 1 computes $\vec{s_1} = \vec{q_1} \cdot A$, server 2 computes $\vec{s_2} = \vec{q_2} \cdot A$ and PrivateMart's server computes $\vec{s_3} = \vec{q_3} \cdot A$, following the usual matrix multiplication operation with modulo 2 additions, as discussed earlier.

(f) Servers 1, 2 and PrivateMart's server send back $\vec{s_1}$, $\vec{s_2}$ and $\vec{s_3}$ back to the user respectively.

(g) The user computes $\vec{s} = \vec{s_1} \oplus \vec{s_2} \oplus \vec{s_3}$.

3. [3 marks] Show that at the end of the protocol the user obtains row c of matrix A. That is, $\vec{s} = \vec{q} \cdot A = A_c$. You may express you proof using words or mathematical expressions, but be sure to give a detailed explanation.
4. [2 marks] In the above solution provided by PrivateMart, do the servers learn any information about the desired row? If yes, explain what they learned, and if not, explain why not.
5. [1 marks] What is the total number of matrix elements that are uploaded and downloaded in iCompany's IT-PIR solution?
6. [1 marks] Suppose $m = 10$. Give a range of (integer) values for n (number of products in the database) where your trivial scheme from part (a) involves communicating (uploading and downloading) *more* matrix elements than PrivateMart's IT-PIR solution above.