UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2023**

Yousra Aafer
Urs Hengartner

ASSIGNMENT 3

Assignment release: **Friday, November 3, 2023**
Milestone due date: **Friday, November 17, 2023 at *3:00 pm***
Assignment due date: **Friday, December 1, 2023 at *3:00 pm***

Total Marks: 84 (36 written + 48 programming)
Bonus Marks: 0 (no bonus marks)

Written Response TA: Andy Yu
Programming TA: Ruizhe Wang

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

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).

Written Response Questions [36 marks]

Q1: Two-time Pad [4 marks]

In the One-time Pad (OTP) cryptosystem, the secret key must be truly random and should never be reused. Let C denote the ciphertext, P denote the plaintext, and K denote the key. OTP encryption is defined as $C = P \bigoplus K$, where $\bigoplus$ is the bitwise XOR operator and the key has the bit length l as both the ciphertext and the plaintext.

The following subquestions will help you understand why the two-time pad is insecure. Suppose that we have two ciphertexts C_1 and C_2 using the same key K (i.e., "Two-time Pad"), where their plaintexts P_1 and P_2 as well as the key are **unknown** to us.

1. [2 marks] Please describe how to obtain $P_1 \bigoplus P_2$. Be sure to provide a brief proof.
2. [2 marks] Suppose both plaintexts are meaningful texts that contain the ASCII characters from only 52 English letters (i.e., "A-Z a-z") and space. Assume you have no other information about the plaintexts. Please describe how to recover the plaintexts P_1 and P_2.

Q2: Relay Selection in Anonymity Networks [15 marks]

Anonymity systems such as the Tor network forward traffic via a sequence of relays. In context of Tor:

- We refer to one such sequence of relays used to forward traffic as a *circuit*.
- We refer to the first and last relay in a circuit as the *guard relay* and the *exit relay*, respectively.
- All other relays on a circuit are called *middle relays*.

When users frequently rely on an anonymous communication system, it is important to consider if and how they change the relays in their circuit/circuits. In the following, we evaluate multiple strategies for changing relays.

(a) [1 mark] What happens if an attacker controls the exit, middle, and guard relays of a circuit?

(b) [2 marks] Suppose we make the following assumptions:

- An anonymity network periodically changes all relays of a circuit.
- A certain fraction of relays are malicious (and controlled by one common entity).
- Malicious relays can be chosen as exit, middle, and/or guard relays.

Name two effects of the periodic changes on the user's privacy. (Hint: recall the nymity slide.)

(c) [2 marks] Suppose we make the following assumptions:

- A user connects to the same personal email account every time they use the anonymity network.
- The email provider colludes with the adversary who controls the relays in (b).

In the scenario of (b), how many of the connections to the email account can be linked to the user? Explain.

(d) [4 marks] Assume relays do not fail/disappear from the system, i.e., availability of the chosen circuit is not a problem. Consider an anonymity network that does not change circuits over time. Name one advantage and one disadvantage of this approach in contrast to periodic changes. Explain.

(e) [1 mark] Tor actually chooses the guard relay from a small set of relays (usually 3), chosen when the user first joins the system. The middle and exit relays change periodically and are selected from all available relays. Why is that a sensible approach?

(f) [2 marks] Tor relays do not delay or re-order packages that they forward. How can we take advantage of this fact to link the source and destination of a circuit when controlling only the exit and guard node? Explain.

(g) [3 marks] A dog owner wants to publish the below image anonymously. Explain how Tor Onion/Hidden Services enable anonymous publication (you can use external sources, such as the Tor project homepage).

Q3: Data Privacy [11 marks]

Suppose the university has `Student`, a table of size N, in its database. Below is an excerpt (i.e., not the entire table):

Name	Birthdate	Gender	Postal Code	Grade
Lucy	0819	F	N2R 3M5	68
Bob	0214	M	B7S 1Z9	98
Mallory	0304	F	G3R 4S2	84
Charlie	0222	M	G3R 4S2	71
Annette	0601	F	B7S 1Z9	89
Robert	1111	M	B7S 1Z9	85
Adam	1230	M	G3R 4S2	84
Phoebe	0509	F	N2R 3M5	92
Amelia	0229	F	G3R 4S2	73

Table 1: Part of `Student` Table

To deter people from learning other people's grades, the database is set up to suppress the `Grade` field in the output of queries. That said, users can execute queries of the form

```
SELECT SUM(Grade) FROM Student WHERE ...
```

where queries that match fewer than k or more than $N - k$ (but not all N) records are rejected. We will use $k = floor(\frac{N}{8})$.

(a) [4 marks] Suppose there are roughly the same number of females/males in the database. Now let us make the following assumptions:

- Student names are unique.
- With the exception of Charlie, student names are not known to the attacker.
- The attacker has no additional information about Charlie (not even his gender).

As defined in class, use a tracker attack to design (1) a tracker, (2) a set of *three* queries based on this tracker, and (3) show how to derive/infer Charlie's grade based on the query results. Note that your answer should explicitly point out the tracker. Both the tracker and the three queries need to be of the form

```
SELECT SUM(Grade) FROM Student ...
```

(b) [3 marks] Suppose the university becomes aware of the above tracker attack and forbids `SUM(Grade)` queries. Instead they only allow queries of the form `SELECT COUNT(*) FROM Student WHERE Q`. Note that `Q` may include testing the value of the Grade field. (Again, queries that match fewer than k or more than $N-k$, with the exception of N, records are ruled out.) How would you use queries of this type to learn Natalie's grade (Not shown in Table 1)?

(c) [4 marks] Suppose the university believes the student's postal code should be private, and so wishes to protect this sensitive information in its database. After discovering the flaws in its previous defences, the university decides to anonymize the database.

Name	Birthdate	Gender	Postal Code
*	091*	F	G9Q 3X2
*	113*	F	G9Q 3X2
*	043*	M	H4A 5A6
*	092*	F	Y1R 4J4
*	052*	F	H4A 5A6
*	100*	F	H4A 5A6
*	103*	M	H4A 5A6
*	123*	M	H4A 5A6
*	052*	M	Y1R 4J4
*	121*	M	Y1R 4J4
*	071*	F	G9Q 3X2
*	112*	F	H4A 5A6
*	072*	M	H4A 5A6

Table 2: Part of the `Student` Table (Anonymized)

Declaring the first three columns of the table to be identifiers, the university believes that Table 2 is 3-anonymous. Is the university correct? If so, explain your answer as to why it is

3-anonymous. If not, explain why it is not 3-anonymous and then (1) produce a 3-anonymous table and (2) give the value ℓ for which the table is ℓ-diverse.

Q4: Legal Issues [6 marks]

In 2018, the Librarian of Congress ruled for exemptions to the Digital Millennium Copyright Act (DMCA). You can read the details of the ruling here: `https://www.federalregister.gov/d/2018-23241`. The exemptions outline scenarios in which users can circumvent copy protection mechanisms. Read the ruling and then answer the following questions **in your own words**.

1. [2 marks] There was a petition to expand exemptions for computer programs that operate 3-D printers. Can you briefly state what this petition is about and then explain the justifications behind this petition?

2. [2 marks] The EFF filed a petition to add voice assistant devices to the list of devices allowed to be jailbroken. What was the argument of the opposition against this proposed exemption? If their reasons did not succeed (i.e., the exemption passed), what do you think is a likely response from companies selling such devices?

3. [2 marks] In 2015, the Librarian of Congress had also ruled for exemptions to the DMCA: `http://federalregister.gov/a/2015-27212`. However, a major shortcoming of this 2015 DMCA ruling was that security researchers were restricted to circumventing copyright mechanisms for research purposes only to motorized land vehicles, medical devices designed for implantation, and devices designed for individual consumers.

   In 2018, the ruling removed the device restriction. Does this allow a security researcher to now hack into say the computer system of some laboratory for the purpose of "good-faith security research"? If yes, provide two reasons from the ruling to justify your answer. If no, state two reasons from the ruling which prevents that.

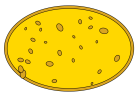
Programming Questions [48 marks]

Note: This assignment is long because it includes a lot of text to *help* you through it.[1]

> 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

Instant messaging (IM) apps have revolutionized the way we communicate, enabling real-time conversations across vast distances with the tap of a button. However, with the ease of communication comes the challenge of ensuring privacy and security. In this assignment, we will collaborate to develop a "strong encryption" protocol for transmitting secure messages on an insecure IM platform: HashBrowns. We ensure that only the sender and recipient can read the messages, keeping them safe from prying eyes, be it hackers, governments, or even the service providers themselves.



HashBrowns

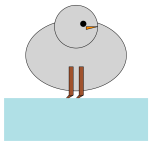
We will begin with the transmission of plaintext JSON messages and we will incrementally introduce security enhancements as we progress.

Each section will outline a distinct protocol for message transmission across the network.

For each question, you will use the `libsodium`[2] cryptography library in a language of your choice to send a message through a web API.

[1] Figures and story texts in this section are generated by GPT-4.
[2] `https://libsodium.org/`

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



Gooser

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

The assignment website is the IM platform you are going to interact with, and also 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. 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[3] to find an interface for your language), you will need to limit your language choice as not all `libsodium` language

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

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
- 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: Plaintext JSON Communication [10 marks]

In this first question, we will completely ignore possible threats and instead focus on only communicating with your parter, Gooser.

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 Gooser 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 Gooser, 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 `Gooser`. The `msg` key specifies the message to send, encoded using `base64`.

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

```
{"api_token": "3158a1a33bbc...9bc9f76f",
"recipient": "Gooser", "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 Gooser 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`.

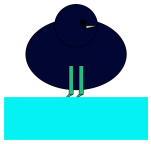
The response to your request is a JSON-encoded array with all of the messages that have been sent to you. Each array element is an object with `msg_id`, `sender`, and `msg` keys. The `msg_id` is a unique number that identifies the message. The `sender` value is the username that sent the message to you.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

Decode the message that Gooser sent you. The message should contain recognizable English words (translated from "honk honk honk"). **The messages from Gooser are meaningless to you and randomly generated.** We use English words so that it is obvious when your code is correct, but the words themselves are completely random as Gooser does not speak English and you don't understand their language.

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 Gooser sent to you in the "Question 1" section to receive your mark.

Question 2: Communicate with Pre-shared Key [10 marks]

Goosper, a friend of Gooser, harbors jealousy over your bond with Gooser. Taking the advantage of being a Bird Intelligence Agency (BIA) agent, Goosper contacts with HashBrowns privately, trying to pursuade them to divulge your communications with Gooser.



Goosper

Bird Intelligence Agency (BIA)

Luckily, Goosper cannot present an official order and HashBrowns decides to disclose this threat to you, You are also warned that if one day an order is presented, they need to send all your message histories to Goosper without noticing you. Observing this menace, you meet with Gooser offline and decide to send encrypted messages using secret-key encryption from now on. You and Gooser created a secret key together.

In this part, you will extend your code from question 1 to send encrypted messages using secret-key encryption. 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": "Gooser", "msg":
  base64encode(concat(nonce, secretbox(plaintext, nonce, key)))}
```

To receive marks for this part, send an encrypted message to `Gooser` 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]

Gooser 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: Pre-shared Password Encryption [8 marks]

"I cannot remember the key !!!", exclaimed your partner Gooser. The intricacies of a 32-byte key seem to elude Gooser's memory. Considering this, you suggest a change in approach: instead of sharing and storing complex keys, both of you should simply exchange and remember a password. The secret keys can be derived from robust passwords using iterated hash functions. This method is not only more user-friendly but also easier to recall. Gooser remains skeptical, prompting you to provide a demonstrative example to validate your proposal.

Question 3 part 1: send a message [6 marks]

Using the exact same format as in question 2 part 1, send a message to Gooser using the following web API page:

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

Here, `psp` stands for "pre-shared password". The only thing to change is that, instead of passing in a secret key directly, you must instead iteratively hash a pre-shared password to derive the key.

Visit the "Question Data" page and retrieve the password and salt from the "Question 3" section. Use the Argon2id password hashing functionality of `libsodium` to derive the secret key from this password and salt. Argon2id is the default password hashing algorithm in `libsodium`. In the C library, the function that accomplishes this task is called `crypto_pwhash_argon2id`. Libraries for other languages might not have an explicit argon2id identifier in the function name (since it is the default); check the library documentation to make sure you're using Argon2id. In case there are different versions of Argon2id, use version 1.3. This function performs a large number of cryptographic hashes and memory-hard operations[4] to derive the secret key; this procedure greatly increases the amount of time required to guess the password by brute force.

The iterative hashing function also takes as input the number of computations to perform and the maximum amount of RAM to use. You should use the "`INTERACTIVE`" constants provided by `libsodium` to configure these values. If your language binding does not expose these constants (e.g., the `nacl` module for Python), then you will find the values to use in the "Question 3" section of the "Question Data" page.

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

Check your inbox using this web API page:

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

Derive the secret key from the password, decrypt the message, and enter the plaintext that Gooser sent you in the "Question 3" section of the "Question Data" page.

[4] These operations are intentionally designed to be difficult to perform on devices with small amounts of memory, such as custom password-cracking hardware.

Question 4: Digital Signatures [10 marks]

Anyone enrolled in CS 458/658 understands the perils of jotting down passwords. However, Gooser, having never taken the course, naively records the password in a notebook. Predictably, Goosper discovers this and, in an attempt to sever your communication with Gooser, whisks Gooser away on a sudden journey. The old key is no longer secure and you cannot exchange a new key with Gooser physically. You seem to lose the ability to communicate with Gooser, forever.

Yet, just a day after Gooser's departure, you receive a mail from it with a note suggesting the use of public-key cryptography, also known as asymmetric cryptography, for future conversations.

"Goosper monitors each of my messages, so transmitting the key online isn't an option," warns Gooser. "Moreover, you should always authenticate my online identity before trusting any message. It could be a ruse by Goosper!" You sense the gravity of the predicament but feel at a loss for immediate action, prompting you to read further. "To begin, let's authenticate that each message indeed comes from me," Gooser stresses. "I've securely put my signature on HashBrowns. Ensure you verify my messages against it every time."

Drawn by an unshakable sense of caution, you find yourself compulsively verifying your partner's certificate. But then, as if struck by a sudden stroke of insight, a new thought emerges, piercing the calm. The messages you are going to send - what if they are tampered? The very idea sends a ripple of urgency down your spine.

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

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 Gooser, 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 Gooser 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. Gooser will be able to verify the authenticity of your message using your previously uploaded verification key.

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

Now it goes to the last step. 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 Gooser'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 Gooser 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 Gooser'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 Gooser'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 Gooser'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, `Gooser`). 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 Gooser 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 Gooser, 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 Gooser. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. This function takes as input Gooser'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": "Gooser",
                        "msg": base64encode(
concat(nonce, box(plaintext, nonce, Gooser_public, your_secret))
                                )}
```

Finally, send the message to Gooser 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 Gooser 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 Gooser, you will need your secret key and Gooser'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.