UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

**CS 458/658** **Computer Security and Privacy** **Spring 2017**

**Ian McKillop, Stefanie Roos**

ASSIGNMENT 3

Assignment due date: **Friday, July 21, 2017 *3*:00 pm**

**Total Marks:** 64 (+6 bonus)
**Written Response TA:** Kshitij Jain (k22jain@uwaterloo.ca)
**Office hours:** Monday 2:00–3:00pm, DC 2569
**Programming TA:** Nik Unger (njunger@uwaterloo.ca)
**Office hours:** Tuesday 1:30–2:30pm, DC 3332

Please use LEARN for general discussion, emails or office hours for personal questions. DO NOT POST PARTIAL SOLUTIONS ON LEARN!

# Written Response Questions [32 marks]

**Note:** Please ensure that your written answers use complete and grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as the content.

## Question 1: GnuPG [8 marks]

A GnuPG public key for k22jain@uwaterloo.ca is provided along with the assignment on the course website (**k22jain.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 for yourself. Use the RSA and RSA algorithm option, your real name, and an email address of your-userid@uwaterloo.ca. Export this key using ASCII armor into a file called **key.asc**. (Note: older versions of GnuPG might not have the RSA and RSA algorithm 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 k22jain@uwaterloo.ca key. Its true fingerprint can be found on the course website (**k22jain.fpr**). Export your signed version of the k22jain key into a file called **k22jain-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 k22jain 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.

## Question 2: Length Extension Attack [10 marks]

Give answer to all the sub-parts briefly. Provide any additional sources you consulted for answering the questions.

(a) [3 marks] Read Thai Duong and Juliano Rizzo's attack against Flickr's API[1]. Briefly explain their exploit, specifically discussing how they used the structure of MD5 for the exploit.

(b) [4 marks] Is SHA-256 exploitable using a similar length extension attack? Based on the exploit, what can you say about "SHA-256 being a random oracle[2]"?

(c) [3 marks] What is a hash-based message authentication code[3] (HMAC)? How do HMACs help to mitigate length extension attacks?

## Question 3: Relay Selection in Anonymity Networks [14 marks]

Anonymity systems such as the Tor network forward traffic via a sequence of relays. In the 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* and the *exit relay*, respectively. All other relays on a circuit are called *middle relays*. When users rely on an anonymous communication system frequently, it is important to consider if and how they change the relays in their circuit or circuits. In the following, we evaluate multiple strategies for changing relays.

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

(b) [2 marks] Assume that an anonymity network changes all relays of a circuit periodically. Furthermore, assume that a certain fraction of relays are malicious (and controlled by one common entity) and that malicious relays can be chosen as guard, middle, and exit relays. Name two effects of the periodic changes on the user's privacy. (Hint: remember the -nymity slide.)

[1] http://netifera.com/research/flickr_api_signature_forgery.pdf

[2] https://blog.cryptographyengineering.com/2011/09/29/what-is-random-oracle-model-and-why-3/

[3] https://tools.ietf.org/html/rfc2104

(c) [2 marks] Assume that a user connects to the same email account every time they use the anonymity network. Assume that the email provider colludes with the adversary who controls 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] 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. Assume that relays do not fail/disappear from the system, so that availability of the chosen circuit is not a problem.

(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 re-order or delay packages they forward. Explain how this can be used to link the source and destination of a circuit when controlling only the guard and the exit node.

(g) [2 marks] The panda falling off the mattress in the picture below wants to publish the picture anonymously. Explain shortly how Tor Hidden Services enable anonymous publication (you can use external sources, such as the Tor project homepage).

# Programming Question [32 marks + 6 bonus]

> 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

> If the challenges of real-time interception threaten to leave us in the dark, encryption threatens to lead all of us to a very dark place.
>
> JAMES COMEY
> *Former FBI director*

> Security is more than encryption, of course. But encryption is a critical component of security. You use strong encryption every day, and our Internet-laced world would be a far riskier place if you didn't.
>
> BRUCE SCHNEIER
> *Cryptographer & security writer*

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`[4] 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 *Jessie* in order to confirm that your code is correct. However, if you complete all of the programming part questions, you will be able to use your code to send secure messages to other students who have also completed all questions (or to the TA).

**Assignment website:** `https://whoomp.cs.uwaterloo.ca/458a3`

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

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

## Choosing a programming language

Before beginning the questions, you should choose 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.

However, you will need to use `libsodium` to complete the assignment. While `libsodium` is available for dozens of programming languages, you will need to limit your language choice to the available options. Check the list of language bindings[5] to find an interface for your language.

Not all `libsodium` language bindings support all of the features needed for this assignment. While it is not important to understand the meaning of the cryptographic terms, you should quickly check the documentation for your language to ensure that it gives you access to the following `libsodium` 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
- "Password hashing": key derivation function (KDF) using Scrypt with Salsa20/8 and SHA-256
- "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 any third-party libraries and code. However, if you copy 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://github.com/pyca/pynacl`) to wrap `libsodium`. The `base64`, `json`, and `requests` modules from the standard library work well for interacting with the web API. The box and secret box implementations include nonces in the ciphertexts, so you do not need to manually concatenate them.

[5] `https://download.libsodium.org/doc/bindings_for_other_languages/`

- **PHP**: This language works well if you are already familiar with it. Use the `libsodium` extension (https://github.com/jedisct1/libsodium-php) for cryptography. 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. The `libsodium-php` extension is relatively new, so you might have to compile the module yourself if you are not using a very recent operating system. On Debian or Ubuntu stable releases, you might need to install the `php-dev` package and manually include the compiled `libsodium.so` extension in your CLI `php.ini`.
- **Java**: This language is a reasonable choice if you are comfortable using it, but getting `libsodium` to work can be tricky. The `libsodium-jna` binding (https://github.com/muquit/libsodium-jna) works, but it is incomplete. If you choose to use Java, check LEARN for modified bindings containing all of the functions that you will need. 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`.
- **JavaScript**: The simplest way to solve the assignment in JavaScript is to use Node.JS with the `libsodium.js` binding (https://github.com/jedisct1/libsodium.js). Unfortunately, the method signatures in `libsodium.js` are slightly different than the C library, and the wrapper is poorly documented; you may need to look at the prototypes in `wrapper/symbols/` to identify the inputs and outputs. You should use the `dist/modules-sumo` package (not the default), since it includes the required hashing functions. `libsodium.js` includes helper functions for converting between hexadecimal and binary. The standard library includes the other encoding functions you will need: `atob` and `btoa` for `base64`, and the `JSON` object for JSON processing. Be wary of string encoding: you may need to use the `from_string` and `to_string` functions in certain situations.
- **Go**: We only recommend using Go for this assignment if you are already familiar with the language and its recent vendoring tools (e.g., `govendor`). The best `libsodium` binding for Go is `libsodium-go` (https://github.com/GoKillers/libsodium-go). However, the binding is incomplete and the latest tagged release (`v0.4-beta`) contains a bug that makes questions 5, 6, and 7 impossible to solve. At the time of this writing, the latest `master` branch also fails to compile with `libsodium` 1.0.11. Consequently, you should use `govendor` to install only the following subpackages from `master`: `cryptobox`, `cryptogenerichash`, `cryptosecretbox`, and `cryptosign`. You will also need to use the Go cryptography library's implementation of Scrypt (`golang.org/x/crypto/scrypt`) to compute password hashes, since `libsodium-go` does not include a binding for this feature. Note that the official Scrypt library expresses its configuration in a different way than `libsodium`. You should use $N = 16384$, $r = 8$, and $p = 1$ as parameters, which corresponds to the "`INTERACTIVE`" hardness configuration in `libsodium`. All other features are provided by the standard library.
- **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.haxx.se/libcurl/) for submitting API requests, `Jansson` (http://www.digip.org/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 and libraries will be made available on the Ugsters in case you do not have access to a personal development computer. We will install some of the languages shortly after the assignment is released. Check the LEARN discussion forum to see which languages are available.

## `libsodium` documentation

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

https://libsodium.org/doc/

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.

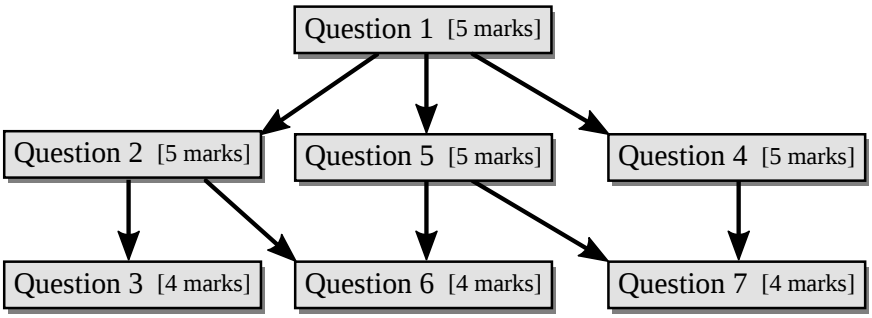
In the past, the website has been unavailable for extended periods of time. If the documentation site is down, you can access an offline mirror of the PDF documentation on LEARN.

## Question Dependencies

Every question in the programming part takes place in an isolated environment; no data that you send or receive in one question will appear in another question. Some of the later questions use

techniques that are introduced in earlier questions, so you may find that it is useful to copy and/or reuse code. However, when submitting your assignment, **you must submit code for all questions**, so do not overwrite your code for previous questions!

It is generally advisable to solve the questions in order. However, if you get stuck and want to move on to a later question, you should know that there are several "dependencies" between questions (i.e., solving some questions essentially involves solving previous ones as a component). The following graph indicates question dependencies:




## Question 1: Using the API [5 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 [3 marks]

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

- URL: `https://whoomp.cs.uwaterloo.ca/458a3/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 Jessie, your JSON object should also contain `to` and `message` keys. The `to` key specifies the username for the recipient of your message; this should be set to `jessie`. The `message` key specifies the message to send, encoded using `base64`.

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

```
{"api_token": "3158a1a33bbc...9bc9f76f",
"to": "jessie", "message": "SGVsbG8sIHdvcmxkIQ=="}
```

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 [2 marks]

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

`https://whoomp.cs.uwaterloo.ca/458a3/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 `id`, `from`, and `message` keys. The `id` is a unique number that identifies the message. The `from` value is the username that sent the message to you. The `message` value contains the `base64`-encoded message.

Decode the message that Jessie sent you. The message should contain recognizable English words. **The messages from Jessie 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 also allows you to submit answers to certain questions. Enter the decoded message that Jessie sent to you in the "Question 1" section to receive your mark.

## Question 2: Pre-shared Key Encryption [5 marks]

In this part, you will extend your code from question 1 to encrypt messages using secret-key encryption. For now, we will assume that you and Jessie have somehow securely shared a secret key at some point in the past. 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 [3 marks]

Send a request to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/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 `message` 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 `libsodium`

documentation contains examples). To generate the message, you should `base64` encode a concatenation of the nonce with 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", "to": "jessie", "message":
  base64encode(concat(nonce, secretbox(plaintext, nonce, key)))}
```

To receive marks for this part, send an encrypted message to `jessie` 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 [2 marks]**

Jessie 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://whoomp.cs.uwaterloo.ca/458a3/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 [4 marks]

Securely sharing 32-byte keys is not very convenient. It is slightly more convenient to securely share passwords, but passwords themselves cannot be used for secret-key encryption. However, we can derive secret keys from reasonably secure passwords by using iterated hash functions, as discussed in class in the context of web applications.

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

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

`https://whoomp.cs.uwaterloo.ca/458a3/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 Scrypt password hashing functionality of `libsodium` to derive the secret key from this password and salt. In the C library, the function that accomplishes this task is called `crypto_pwhash_scryptsalsa208sha256`. This function performs a large number of cryptographic hashes and memory-hard operations[6] 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 [1 mark]

Check your inbox using this web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/psp/inbox`

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

## Question 4: Digital Signatures [5 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 Jessie.

#### Question 4 part 1: upload a public key [2 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 public keys must then be distributed somehow. For this assignment, the web API will act as a *public key directory*: everyone can upload a public key, and request the public keys that have been uploaded by other users.

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

`libsodium` implements public-key cryptography for digital signatures as part of its `sign` functions. Before sending a message to Jessie, 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 signing key somewhere (e.g., a file), because you will need it for the next part. To receive marks for this part, upload the public verification key to the server by sending a `POST` request with the usual headers to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/signed/set-key`

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

```
{"api_token": "3158a1a33bbc...9bc9f76f",
 "public_key": "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 public key.

**Question 4 part 2: send a message [3 marks]**

Now that you have uploaded a public 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 Jessie by sending a request to the following web API page in the usual way:

`https://whoomp.cs.uwaterloo.ca/458a3/api/signed/send`

The `message` 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. Jessie will be able to verify the authenticity of your message using your previously uploaded public verification key.

## Question 5: Public-Key Authenticated Encryption [5 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.

The `libsodium` library refers to an authenticated public-key ciphertext as a "box" (in contrast to the "secret box" used in questions 2 and 3). Internally, `libsodium` performs a *key exchange*

between your secret key and Jessie'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 [2 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 Jessie 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 for that key, then the server would be able to decrypt it; it could even re-encrypt it under Jessie'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 that you somehow verify that you received Jessie's actual public key. This is a very difficult 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 Jessie'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://whoomp.cs.uwaterloo.ca/458a3/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, `jessie`). The server's response will be a JSON object containing `user` (the requested username) and `public_key`, a `base64` encoding of the user's public 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 Jessie securely gave to you. To get the marks for this part, enter the hash of the public key, in hexadecimal encoding, into the "Question 5" section of the "Question Data" page.

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

Before sending a message to Jessie, 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 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://whoomp.cs.uwaterloo.ca/458a3/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 Jessie. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. This function takes as input Jessie's public key (which you downloaded in the previous part), 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", "to": "jessie",
                "message": base64encode(
concat(nonce, box(plaintext, nonce, jessie_public, your_secret))
                )}
```

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

`https://whoomp.cs.uwaterloo.ca/458a3/api/pke/send`

**Question 5 part 3: receive a message [1 mark]**

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

`https://whoomp.cs.uwaterloo.ca/458a3/api/pke/inbox`

To decrypt the message from Jessie, you will need your secret key and Jessie'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.

## Question 6: Government Surveillance [4 marks]

The government has decided that they must be able to decrypt all secure messages sent between you and Jessie through the web server. They have devised a new protocol that will protect the contents of your message from everyone except you, Jessie, and them. In the new protocol, you will use hybrid encryption: the message will be encrypted with secret-key encryption, and then the secret key will be encrypted using public-key encryption. Normally, you would encrypt the secret key using only Jessie's public key. In this new protocol, you will *also* encrypt the secret key using the government's public key. This way, both Jessie and the government will be able to use their secret key to decrypt the message.

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

Begin by generating and uploading a public key in the exact same way as for question 5 part 2, except using the following web API page:

```
https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/set-key
```

Next, download Jessie's public key in the exact same way as for question 5 part 1, except using the following web API page:

```
https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/get-key
```

Visit the "Question Data" page and obtain the government's public key (in `base64` encoding) from the "Question 6" section.

Now it is time to create your encrypted message for Jessie. Do this by following these steps using the appropriate functions for your language:

1. Generate a random key, called the *message key*, for "secret box" encryption.
2. Encrypt the plaintext with the message key using the same technique as question 2. The resulting ciphertext is called the *message ciphertext*. The nonce for this ciphertext is called the *message nonce*.
3. Encrypt the message key with Jessie's public key and your secret key using the same technique as question 5. The resulting ciphertext is called the *recipient ciphertext*, and its nonce is called the *recipient nonce*.
4. Encrypt the message key with the government's public key and your secret key using the same technique as question 5. The resulting ciphertext is called the *government ciphertext*, and its nonce is called the *government nonce*.

Now construct the message that you will send to the web API. The message should be the concatenation of these values, in order:

1. The recipient nonce
2. The recipient ciphertext (including encrypted message key and a MAC)
3. The government nonce
4. The government ciphertext (including encrypted message key and a MAC)
5. The message nonce
6. The message ciphertext (including encrypted plaintext and a MAC)

Send this message using `base64` encoding to Jessie in the usual way with a request to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/send`

**Question 6 part 2: receive a message [1 mark]**

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

`https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/inbox`

To decrypt the message from Jessie, use Jessie's public key and your secret key to decrypt the ciphertext produced for you (i.e., the recipient ciphertext, *not* the government ciphertext). This will give you the message key. You can completely ignore the government ciphertext. Use the message key to decrypt the message ciphertext and recover the plaintext. Provide the decrypted plaintext in the "Question 6" section of the "Question Data" page.

## Question 7: Forward Secrecy [4 marks]

The protocols in all of the previous questions share a problem: if an eavesdropper passively records all of the encrypted messages and later steals one of the secret keys, then they can retroactively decrypt any messages that they previously stored. If we stop this from happening, then we achieve *forward secrecy* (sometimes called *perfect forward secrecy*).

The "trick" for forward secrecy is to encrypt messages using temporary secret keys and authenticate them using long-term secret keys. Stealing long-term secret keys that are only used for authentication does not affect previous conversations, since they have already concluded. It is also generally more difficult to steal secret keys that are erased quickly than it is to steal secret keys that must be kept around for a long time.

The protocol for this question shares aspects of the signed messages protocol in question 4, and the public-key encryption in question 5. However, in this question, every user will upload two public

keys to the server: a public *identity verification key*, and a *signed prekey*.[7] The identity verification key is used for authentication via digital signatures. It is produced in the same way as in question 4. The signed prekey is used for encryption. It is produced in the same way as in question 5, with the exception that it is also signed by the identity verification key.

Unlike identity verification keys, signed prekeys are meant to be changed regularly. Once both the sender and receiver of a message have deleted their old signed prekeys, the message cannot be retroactively decrypted by stealing secret keys.

When sending a message to Jessie in question 5, the "box" was created using Jessie's public key and your secret key, both of which are long-lived. Here, the "box" will be created using Jessie's *signed prekey* and the secret for your *signed prekey*.

### Question 7 part 1: upload a signed prekey [1 mark]

In this first part, you will generate and upload a signed prekey so that others can send messages to you.

The first step is to generate and upload an identity verification key in the same manner as question 4 part 1. Produce a signing and verification key using the equivalent of the C function `crypto_sign_keypair` in your language. Upload the verification key in the usual way (encoded in `base64` within the `public_key` property of a JSON object in the `POST` request) through the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/set-identity-key`

If the upload was successful, you'll receive a `200` HTTP status code in response.

Next, generate a prekey using the same technique as question 5 part 2. Since prekeys will be used for public-key encryption, you should generate your prekey using the equivalent of the C function `crypto_box_keypair` in your language.

To produce a signed prekey, use your identity signing key to sign the public key in the same way as in question 4 part 1. Use the equivalent of the C function `crypto_sign` in "combined mode" to produce the signed prekey. Finally, `base64` encode the signed prekey and send it in the `public_key` property of a JSON object to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/set-signed-prekey`

If you receive a `200` HTTP status code in response, then you have received the mark for this part.

[7] A "prekey" is just an ordinary public key with a short lifetime. This terminology was introduced by the secure messaging application called Signal.

**Question 7 part 2: send a message [2 marks]**

Now that you have uploaded a signed prekey, you are ready to send a message to Jessie. Download Jessie's identity verification key sending a request containing a JSON object with `jessie` in the `user` key to this web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-identity-key`

Your request body should look similar to this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "user": "jessie"}
```

You should receive the `base64`-encoded identity verification key in the `public_key` key of a JSON object in the response.

Using the exact same technique, request Jessie's signed prekey by sending a request to this web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-signed-prekey`

This time, you should receive Jessie's signed prekey, encoded using `base64`, in the `public_key` property of the JSON object. You should now verify the signature on this prekey, and extract the public key, by using the equivalent of the C function `crypto_sign_open` in your language. If the signature is valid, this function will return the public key to use for encryption.

Finally, you can encrypt a message to send to Jessie. Encrypt the plaintext using Jessie's prekey as the public key, and the secret key associated with your prekey that you generated in part 1. Use these keys for public-key authentication using the equivalent of the C function `crypto_box_easy` in your language. This is the same public-key authenticated encryption technique that you used in question 5 part 2. As before, you should include a newly generated random nonce in your message.

Send the nonce and the ciphertext to Jessie by sending a request in the usual manner to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/send`

Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "to": "jessie",
                    "message": base64encode(
concat(nonce, box(plaintext, nonce, jessie_prekey, your_prekey_secret))
                    )}
```

If Jessie is able to successfully decrypt your message, you will receive a `200` HTTP status code in the response indicating that you have been awarded the marks for this part.

**Question 7 part 3: receive a message [1 mark]**

Finally, you will need to receive a message sent to you by Jessie using the same encryption scheme as the previous part. Check your inbox in the usual manner using the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/inbox`

When you receive a message, you should look up the identity verification key and signed prekey of the sender (if you have not done so already). You can do this by submitting requests to `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-identity-key` and `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-signed-prekey` as described in part 2. Using Jessie's prekey and the secret key associated with your prekey, decrypt the ciphertext using the equivalent of the C function `crypto_box_open_easy` in your language. This is the same decryption function that was used in question 5 part 3.

Once you have decrypted the message sent to you by Jessie, enter the message in the "Question 7" section of the "Question Data" page to receive the mark for this part.

## Freedom Environment

If you successfully complete every question in the programming part, then you will be given access to the "freedom" environment. Here, you can use the code that you wrote for question 7 to communicate with other students who have also completed all of the parts, assuming that you know their WatIAM username and they have uploaded keys in the environment. The programming part TA is also available in this environment, with username `njunger`. To communicate, use the following web API pages:

`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/get-identity-key`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/set-identity-key`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/get-signed-prekey`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/set-signed-prekey`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/send`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/inbox`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/`*`message_id`*`/delete`

To delete messages from your inbox, send a `POST` request in the usual manner to the `delete` web API page listed above. The *`message_id`* in the URL is the `id` value of the message provided in the response to an `inbox` request. A `200` HTTP status code in the response indicates that the message has been removed from your inbox.

## Bonus Question [+6 marks]

Answer ***at most one*** of the following bonus questions (if you answer more than one question, only your first answer will be marked). Note that academic integrity rules still apply to bonus questions; written answers should be your own, and quotes or ideas from others must be properly attributed.

1. Consider the protocol for government surveillance presented in question 6. Is it a good idea for a government to mandate that all encryption works in this way? Provide arguments in favor of this idea *and* arguments against this idea. Your complete answer should be 6–10 sentences.
2. The academic community considers the Signal app[8] from Open Whisper Systems to be one of the most secure communication apps available to the public today. Investigate the design of the Signal protocol. Point out three distinct features of the Signal protocol (1-2 sentences each) that differ from the design of the protocol in question 7. For each feature, explain why Signal uses this feature (1-2 sentences each).
3. (*Very difficult*) Consider what can happen if the web server is malicious. Assume that everyone communicates using the protocol from question 7, and that everyone has perfectly verified everyone else's identity verification key using a cryptographic hash (as described in question 4 part 1); in other words, the server cannot lie about a public key uploaded by a user. The server is not allowed to delay or drop messages, although it is allowed to modify them. The server is also not powerful enough to compromise secret keys of honest users. It is still possible for an actively malicious server to perform an attack that causes a recipient to believe a falsehood about a message that they receive. Describe in detail an attack that the server can perform in this situation.

[8] https://whispersystems.org/

# What to hand in

Using the "submit" facility on the student.cs machines, hand in the following files:

**a3.pdf** A PDF file that contains your answers to all written response questions, plus (optionally) your answer to the bonus question for the programming part.

**a3q1.tar** an uncompressed tar file containing key.asc, k22jain-signed.asc and message.asc.

**a3code.tar** an uncompressed tar file containing your code for all parts of the programming question. 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 they were shown on the assignment website.

**Note:** You must include your name, your uWaterloo userid, and your student number at the top of the first page of a3.pdf. Failure to do so will result in a deduction of 5 marks from your final score on this assignment! Also, be sure to "embed all fonts" into your PDF files. Some students' files were unreadable in the past; if we can't read it, we can't mark it.