NATIVE CRYPTOGRAPHY IN THE BROWSER, AN EXPLORATORY APPROACH

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Joey Wilson

December 2016

COMMITTEE MEMBERSHIP

TITLE:	Native Cryptography In the Browser, An Exploratory Approach
AUTHOR:	Joey Wilson
DATE SUBMITTED:	December 2016
COMMITTEE CHAIR:	Zachary Peterson, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Bruce DeBruhl, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Bellardo, Ph.D. Professor of Computer Science

ABSTRACT

Native Cryptography In the Browser, An Exploratory Approach

Joey Wilson

Your abstract goes in here

ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Chapter 2

BACKGROUND

This chapter discusses some of the concepts, tools, and APIs used in development of the messaging application.

2.1 Asynchronous Javascript

2.1.1 Promise API

Javascript executes on a single thread. To keep the user interface responsive during computation most long running computations in javascript are performed asynchronously. The Promise API has come along to assist developers in writing readable code while taking full advantage of asynchronous operations. Lets look a a basic promise example.

```
var asyncAddOne = funciton (num) {
    return new Promise(function (resolve, reject) {
        if (num < 10) {
            resolve(num + 1);
        } else {
            reject( num > 10 );
        }
    });
};
```

```
asyncAddOne(1)
.then( (result) => {
    console.log(result); // will print 2
})
.catch( (error) => {
  console.error(error);
});
```

This example shows the basic pattern of creating an asynchronous function and how to call it. The function asyncAddOne takes a number as a parameter. This number is checked to be less than 10. If that is the case the function returns the number plus 1 by calling the resolve function. If the number is greater than 10 reject is called with the error message num ¿ 10. The anonymous function passed to then will be called if resolve is called with result being the value passed to resolve. The anonymous function passed to catch will be called if reject is called with the value passed to reject as the error value. Now adding one is not a computationally expensive operation so it would be silly to call the asynchronously but the add one code could easily be replaced with intensive mathematical operations or network calls and this structure would function the same.

Now Promises are nice and definitely an improvement over the callback centric patterns of the past but even Promises have their issues as chains can get quite long if there are a lot of asynchronous computations that need to be performed one after another. The next specification for the Javascript language has support for syntax called async await that makes asynchronous operations even easier to work with and reason about.

2.1.2 Aysnc Await

Async await is the latest syntax for writing and calling asynchronous functions in javascript. Let's take a look at the same example used above written with the async await syntax. [2]

```
var asyncAddOne = async function (num) { // Notice the async keyword in
    if (num < 10) {
        return num + 1;
      } else {
        throw new Error( num > 10 );
      }
}


try {
   console.log(await asyncAddOne(1)); // will print 2
} catch (error) {
   console.error(error);
}
```

This example shows how with this new syntax it has become trivial to turn synchronous code into asynchronous code. The only issue with this is right now browsers do not support this syntax. How this issues is fixed is talked about in the Typescript section of this chapter. This syntax was really important to this project though because all of the WebCrypto API function calls are asynchronous, so using this syntax allowed the focus of development to be on getting the cryptography correct and not

handling asynchronous function calls correctly.

2.2 Binary Data in Javascript

Binary data in Javascript is handled by Typed Arrays. This is important because all of the cryptographic operations in the WebCrypto API take Typed Arrays as input parameters and return Typed Arrays as output. Typed Arrays data is represented by Array Buffers. Array Buffers do not provide an interface for reading or writing data. This is left up to views. The different types of views are similar to the C types there is int and uint values from 8 to 64 bits. These allows for array like access into the Array Buffer. [16]

Some of the issues that arise when working with Typed Arrays are portability and mathematical operations. By default a uint view of the data cannot be sent over a network and retrieved on the other side. Since JSON serialization does not provide type information there is no way for the receiving party to know if the array is meant to be a TypedArray or a regular Array. There seems to be lots of different approaches to solve this problem floating across the internet. The solution this project used is in the JWCL section of the Implementation chapter. There is no direct interface into Typed Arrays to do math. For example there is not .add() function. This makes representing a counter with a Typed Array a little bit difficult. The solution this project uses for this is discussed in the OTR section of the Implementation chapter.

Support for binary data is necessary for cryptographic operations to exists in the browser but there are some challenges that developers still face before working with binary data is a seamless process.

2.3 WebCrypto API

The WebCrypto API is still in the editor's draft stage, but browsers have implemented the specification. As of Chrome 53 the support is strong. This project was developed and tested in Chromium 53 and the API functioned as expected. [30] [22]

The WebCrypto API provides a low level interface to cryptographic operations and key management operations. The API is accessible through the window.crypto.subtle global variable in the browser. The API has twelve static functions that provide access to all of the cryptographic operations. These functions are encrypt, decrypt, sign, verify, digest, generateKey, deriveKey, deriveBits, importKey, exportKey, wrapKey, unwrapKey.

The design of the API is to overload these functions. These functions all work on Array Buffers as their input and output types. These functions are also all asynchronous and return Promises. To see how this API functions let's look at and example the crypto.subtle.encrypt function.

```
window.crypto.subtle.encrypt({
        name: "AES-CBC",
        //Don't re-use initialization vectors!
        //Always generate a new iv every time your encrypt!
        iv: window.crypto.getRandomValues(new Uint8Array(16)),
    },
    key, //from generateKey or importKey above
    data //ArrayBuffer of data you want to encrypt
)
```

```
.then(function(encrypted){
    //returns an ArrayBuffer containing the encrypted data
    console.log(new Uint8Array(encrypted));
})
.catch(function(err){
    console.error(err);
});
```

This example was gotten from [4]

This example shows the overloaded nature of the function. This first parameter is a Javascript object containing the options for encrypt. In this case the name was AES-CBC, since CBC mode takes an IV, this is also passed in. If the mode was CTR then a counter would be passed in. A side effect of this is the user of this API needs to have knowledge of the different encryption methods and what parameters they require. This function is also used for public private key encryption as well. In the case of RSA the same function is called with different options. These functions are very overloaded. In the example it is show how the input data and output encrypted data are Array Buffers. Also notice the .then function that is called because the encrypt function returns a promise. All of the other WebCrypto API functions provide a similar overloaded API and their respective call signatures can be seen in the specification.

The WebCrypto is a great start to bringing cryptographic operations natively to the browser.

2.4 Typescript

Typescript is a tool developed by Microsoft to build Javascript that scales [15] Typescript is a strict superset of Javascript that allows for optional type checking and compilation to plain javascript. The strict superset feature means that any valid Javascript is valid Typescript, a great feature for legacy code bases. If used, the optional types allow the Typescript compiler to statically check your code before running it in the browser. This allows errors to be caught at compile time instead of runtime. The compilation allows use of newer Javascript language features that can then be compiled down to older syntax.

This project uses both static typing and compilation support heavily. The type checking helped prevent errors before the code even ran. The compilation allowed the use of the async await syntax in this project which was awesome.

2.5 Websockets

Websockets are a communication protocol specified in RFC 66455. [5] Websockets provide two way communication between a client and a host. This is different from traditional HTTP in which the client sends requests and the host sends responses, each time through a different TCP connection. Websockets allow for data to be pushed to the client which is beneficial from both a client and host point of view.

For the client the data is always up to data. Prior to the development of websockets, web applications employed different strategies to keep the client viewing the most recent data. The simplest was to have the client manually refresh the page periodically. This is bad as it takes client interaction. A slightly better technique and one that is commonly still used today is long polling. AJAX requests will be sent from the client periodically. These requests will say, hash anything changed, if

so send me the newest data. The works okay in practice but has issues with the fact the data is never truly live. The max wait time for up to date data is the polling interval. To combat this the polling interval could be set really short but then the network use of the client is high. On desktops this wouldnt be the end of the world but on mobile devices this poses a huge problem. So there is a balance between poll time and how up to date the clients need to be. Websockets solve this problem by setting up the two way connection. So the client just has to set up a listener on the open connection and react accordingly to up to date data.

For the hosts this can allow for a bufferless server in some cases. Before websockets the server had to store the data for at least until a client asked for it. This could be a really short amount of time depending on the polling rate but the infrastructure and code to buffer data still has to exist. Websockets solve this problem because servers can send out changes as soon as they happen without any buffered storage. This makes servers simpler to develop and deploy.

Websockets have improved the experience for users on the internet while making technical problems easier to solve.

2.6 Flask

Flask is a micro web framework in Python for building web applications. [25] Flask was used in this project to develop the server side. Flask comes with routing support as well as an extension to handle websocket connections. Flask is great for its simplicity and community. It only takes one file to have a basic web server up and running, and there are tons of developer writing extensions to give Flask more features.

2.7 Json Web Tokens

Json Web Tokens, commonly abbreviated as JWTs, are defined in RFC 7519. [**?**] JWTs can be used as tokens to authenticated users. This project uses JWTs as the authentication token from the server.

JWTs are comprised of three parts, a header, payload, and signature. The header is metadata about the token. The header usually has the name of the algorithm used to generate the signature. The payload is the data that the token holds. So in the case of an authentication token this might hold the username and expiration time of the token. The signature is created on the server. The tokens can be signed by either a symmetric or private key. Once the signature has been created all of the information is base64 encoded and separated by the . character. The website http://jwt.io has a live tool that can be used to examine JWTs.

The use of symmetric vs private keys depends on the architecture of the application that the tokens are used in. In a single server system the server can just sign with a symmetric key and then verify using the same key as there is only one place verification needs to take place. In a distributed system where one authentication server is in place for many different application a public private key approach makes more sense. In this system the private key is kept on the authentication server where the JWTs are created. Then the public keys are shared with all of the application servers which can now verify the signature with the public key.

2.8 Off the Record Messaging

Off the record messaging, commonly abbreviated as OTR, is an encrypted messaging protocol. This section goes over some of the properties of encrypted messaging and then gives and overview of the OTR protocol.

2.8.1 Encrypted Messaging Properties

There are four properties that are important when looking at encrypted messaging systems. They are perfect forward secrecy, repudiability, and forgeability.

Perfect forward secrecy is an extension of the confidentially property of encrypted messaging. Perfect forward secrecy is the property that once a short lived key is discarded there is no information that can be gained about past messages. OTR accomplishes this by using short term keys and forgetting them after use.

Repudiability is the property that no one should be able to prove Alice sent a particular message, whether she did or not. OTR accomplishes this by publishing old mac keys so that anyone could have signed the message.

Forgeability is an extension on repudiability. This is the property that not only do we want Bob and Eve to be unable to prove that Alice sent any given message, we want it to be very obvious that anyone at all could have modified, or even sent it. OTR accomplishes this by publishing old mac keys and using a malleable cipher so that anyone could have signed and created the message.

[3]

2.8.2 Protocol

OTR has two main components to the protocol, key exchange, and data exchange. The key exchange consists of five states in which Alice and Bob exchange information to securely agree upon encryption keys. The data exchange is where the messaging and key rotation takes place. Since this project was an implementation of the protocol the implementation section goes into full detail about each step of the OTR protocol.

[23]

2.9 Trust Model

In this thesis an implementation of a an OTR messaging client is developed. The server side to pass messages between clients is also developed. In this section the trust model betweens clients and between the client and server is discussed.

In cryptographic terms the server is an honest but curious adversary. [26]. This means that the server can be trusted to pass messages correctly but will read the messages if it is possible. In this implementation the server is assumed to be honest in performing two actions. The first is correctly and honestly sharing the long term public keys of the users with each other. In this implementation the server acts as the trusted third party to share public keys. The second is the server is trusted to route messages to the specified recipient honestly. The server however is not trusted to not log or read the messages and because of this OTR works well because of its end to end encryption properties.

2.10 Related Work

This chapter discusses a related protocol, implementation, and Javascript Crypto library.

2.10.1 Signal Protocol

The signal protocol is based on the OTR protocol that is implemented in this thesis. The signal protocol provides improvements over OTR. The improvements over OTR are focused on simplifying deniability, adapting the key exchange for asynchronous transports, and improving the forward secrecy ratchet.

OTR make the assumption or inorder recipient of messages. This is a perfectly fine assumption for some cases and can cause a lot of problems in others. In the asynchronous world of instant messaging in order recipient is not guaranteed. The signal protocol aims to make their protocol function in an asynchronous system by adapting how the key exchange functions. This is done by improving upon OTRs three step DH rachet and making it a two step DH rachet. This allows Alice to use the next ephemeral key without advertising it.

The signal protocol ratched is a combination of the three step rachet used by OTR and a hash based ratchet used by the silent circle messaging protocol. The OTR rachet works by advertising the next DH key in the message and then deleting old DH keys after the new one has been acknowledged. The silent circle ratchet works by continually hashing the encryption key every message and deleting the previous one. The OTR rachet has less than perfect forward secrecy properties but has good future secrecy properties. The silent circle rachet hash perfect forward secrecy properties but terrible future secrecy properties. The signal protocol aims to take the pros of both of these approaches while mitigating the cons. Signal derives a root key, like

silent circle, but then continues to use the use of ephemeral keys but they are now generated from the root key rather than advertise. In the signal protocol description this is not detailed very well but they claim this allows for a two step ratchet will still having good future secrecy properties.

The reason that OTR was chosen to be implemented instead of signal was because of this disclaimer on the Github documentation pages for signal. Better documentation of this algorithm is forthcoming. The below description is lacking details needed for a complete implementation.

[27] [29]

2.10.2 Allo Google Chat

Allo is a messaging application that has recently been released by Google. Allo works like a lot of other messaging application with feature like emojis and group chats as well as allowing users to attach images and videos.

Allos security settings and model is different from other messaging applications though. By default Google stores all chat history. This allows Google to provide an intelligent assistant to help you chat. The intelligent assistant provides common features such as autocomplete but also some really cool ones like suggesting movie times or restaurants if you are talking about them in the conversation. You just have to add @google and the assistant handles the rest. This feature has define positives for user experience but also has negatives in terms of security. For the assistant to work Google must be able to read your messages. If Google can read your messages then so can anyone who has the proper legal request.

Google attempts to combat this in Allo by providing an incognito mode. Incognito mode turns on the Signal Messaging Protocol to provide end to end encrypted messaging. With incognito mode the Google assistant no longer works since Google

cant read your messages. The incognito mode allows for a timer to be set for how long messages will be stored on your device, and this is configurable independently for each conversation which is nice. They do warn you though that the recipient can always take a screenshot of the conversation.

Google Allo is a concrete implementation of an encrypted messaging app similar to the application built in this thesis. While Google has many more messaging features available the idea of encrypted chat is similar and nice to see big companies investing in developing this technology further.

[6] [28]

2.10.3 SJCL

The Stanford Javascript Crypto Library, commonly abbreviated as SJCL, is a crypto library written in Javascript that runs in the browser. This library was developed at Stanford University and was published in the 2009 Annual Computer Security Applications Conference. [26] SJCL was an option for client side encryption prior to the publication of the WebCrypto API. SJCL supports older browsers since it was written over 7 years ago. SJCL is still under active development with the most recent commit to the master branch being a month ago at the time of this writing. [1] Due to these factors its use will probably continue into the future, for applications that target a wide audience and cant rely on WebCrypto API support. SJCL also provides a higher level interface than the WebCrypto API. The SJCL interface was the inspiration for the JWCL interface implemented in this project.

Chapter 3

IMPLEMENTATION

This chapter discusses a chat application implementing the OTR messaging protocol. The implementation was broken into four distinct pieces. There is the application server that provides authentication and message passing functionality. There is the application client that provides the user interface and the application functionality. There is the OTR implementation that provides the cryptographic messaging functionality. There is the JWCL wrapper that make the native Web Crypto API more usable to the OTR implementation. All source code can be found on github at https://github.com/jo-wil/Thesis.

3.1 Application Server

The application server was written in Python. Flask was used for the web application framework. [25] Flask Sockets was used to add websocket support to Flask. [24] Pycrypto was used for the server side crypto library. [11]

The application was written as a single page web application so there is only one url that user visits and that is /. This loads the entire application into the user's browser. To support the single page application there are 3 api endpoints that the client application uses to communicate with the server, /api/login, /api/signup, /socket.

3.1.1 /api/login

This endpoint is used for authentication.

Success Flow

The client sends the username and password JSON encoded to the server. The server then looks up the user record in the database by username. The server then compares the salted hash of the provided password with that of the password stored in the database. On success the server generates a json web token encoding the username and returns the token to the client with status code 200.

Error conditions

There are two points of failure possible in the authentication workflow. These are the user does not exist in the database or the user does exists but has provided an incorrect password. According to OWASPs authentication guidelines An application should respond with a generic error message regardless of whether the user ID or password was incorrect. It should also give no indication to the status of an existing account. [21] This is fulfilled by returning an empty response body and status code 400 in both of these error conditions. This provides no information to the client on whether the failure was the user did not exist or the password was not correct.

3.1.2 /api/signup

This endpoint is used for registration.

Success Flow

The client sends the username and password of the desired account JSON encoded to the server. The server first looks to see if the user already exists in the database. If the username is still available a random 8 byte salt is generated and the users provided password is salted and hashed. A record for this new user is created in the database

with the password field holding the salted and hashed password, a websocket field initialized to null, and a public key field initialized to null. An empty body with the response code 200 is returned to the user indication successful creation of the user.

Error conditions

The signup flow has only one error condition and that is user already exists in the database. In this case the record is not added to the database and an empty response body with the status code 400 is returned.

3.1.3 /socket

This endpoint is used for websocket communication.

Success Flows

This endpoint has two workflows, registration and messaging.

To start the websocket receives the JSON encoded data from the client. The first step is to get the JWT from the token field in the message. Validation is performed on this token before any further action is taken by the server. After successful validation of the token, the action field determines whether this is a registration or messaging interaction with the server.

If the action is equal to register the registration workflow is started. The username stored in the JWT is used to retrieve the user record from the database. The websocket field is set to the current connected websocket instance. This is the purpose of the registration action, to determine what user has connected on what websocket instance so messages can be properly forwarded to the desired recipient rather than broadcasting them. The public key from the data is stored in the public key field.

This is used as the long term public key in the OTR protocol. All the active users are informed of the online status of this newly registered user. This is done by sending update messages to all the connected clients not currently registering. The currently registering client receives a register response with the list of current current users and their online/offline status as well.

If the action is equal to message the messaging workflow is started. In this workflow the to field of the data is checked to determine who to send this message to. The recipients websocket instance is retrieved from the database and the message is forwarded unchanged with the exception of deleting the JWT token from the message, as this is used for authentication with only the server and should remain private.

3.1.4 Error conditions

This endpoint has three points of failure in the two workflows. The first two have to do with authentication. If the user doesnt provide a token the error message auth no token is returned. If the user's token does not validate the error message auth invalid token is returned. The third error is in the messaging workflow. If a user attempts to send a message to a user that is not online the error message unavailable is returned.

3.1.5 Database

The database used in this application is a python dictionary data structure. This database is stored in memory and is wiped every time the server is restarted. This decision was made for simplicity of implementation, development workflow and needing no dependencies.

3.1.6 Hash Function

The function used was PBKDF2 with the provided parameters of 10,000 iterations, 16 byte generated key size, and SHA256 as the internal hash function [10][7] These parameters were chosen based on RFC 2898s recommendation. In this RFC they recommend a minimum of 1000 iterations and 8 bytes for the salt. 10,000 iterations were chosen as it did not degrade user experience but it does add 10x computation to each hash for an attacker from the minimum recommended value. An 8 byte salt was chosen in conformity with the recommendation. SHA256 was chosen as the hash function as OWASPs guide on using PBKDF2 says that SHA1 is acceptable for now but a more complex hash function may be needed in the future so the decision was made to use a more complex hash function now. [20]

3.1.7 JWT Implementation

For the JSON Web Tokens (JWT), RFC 7519 was followed to implement basic functionality. [8] Two functions were implemted to provide this functionality, encode and decode.

Encode

To create the token the header which holds the algorithm name used to HMAC, in our case SHA256 and the type which is set as JWT is base64 url safe encoded. Then the payload which is a parameter to this function is base64 url safe encoded as well. The concatenation of the header and payload is then HMACd using SHA256 as the hash function and a randomly generated server side key as the signing key. The header, payload, and signature are concatenated by using the . character as the delimiter. This token is then returned to the caller.

Decode

In decode split is run on the . character to get a list of the three parts of our token, header, payload, and signature. First all the values are base64 decoded so that the raw data is available. The signature is then verified by recomputing the HMAC using the same server side key and making sure it is the same as the provided signature. On success the payload is returned to the caller of the function. On failure null is returned to the caller signifying failure of the decoding process.

Since this application is running as one instance and one service, symmetric key signing with the HMAC function was used.

3.2 Application Client

The application client was written in Typescript and compiled targeting ES6 syntax. The application has no dependencies. The client was built as a single page application so after initial page load the only communication with the server is through the above APIs that the server provides. Since these are all JSON encoded messages the application handles all of the UI creation and updates.

There are two states the application can be in login and chat. On page load an instance of the otr implementation class is constructed and placed it in the global data store. A check of the global data store for a JWT is then done. If a token exists chat is loaded, otherwise login is loaded.

The login state displays a login form prompting the user for a username and password combination. The user can enter their credentials and attempt to login.

On successful authentication with the server, the response body contains a JWT and the status code is 200. In this case the username and token are then stored to the global data store. After this is completed a long term public private key pair are

generated to be used by the OTR protocol. This key pair is also stored in the global data store. After this is complete the application shifts to the chat state.

On authentication failure, which is determined by a response code other than 200, a response to the user is generated by placing the message Invalid username or password. below the login form.

The chat state displays a simple chat interface. The user can view the available contacts and their status, the user can then send messages to these contacts.

A lot happens when the chat state is loaded.

First a websocket connection with the server is opened. When this connection is successfully opened a register message is sent to the server containing the JWT and the public part of the public private key pair.

Then a listener is added to the websocket instance to be run on any received messages. This listener first parses the JSON data received from the socket. The first check is for an error message in the data, if this exists the error message is displayed to the user and the function is returned from. If there is no error the action field of the message is looked at.

There are three actions the server can respond with register, update, and message.

Register and update are actually synonymous for the actions they provide as of now, but the decision was made to keep them as separate actions since they really are distinct behaviors and future work may need support of separate actions for so this was built in with that in mind. The action they do provide is to update the contact list. They receive a contact list from the server, first this list is stored in the global data store. After this is it displayed to the UI showing the user which users exists in the system and their online/offline status.

The message action is actually a wrapper around the OTR receive function. This

action just passes the message along to the OTR receive. After the OTR receive function returns the there is a check if the message contains any plaintext. If this is the case the message is displayed to the user interface.

When the user goes to send a message they type in the desired recipient and the message text. A message object is created that has the fields, action, token, from, to, and text. The action is set to message, token to the JWT, from to the sender's username, to is set to the provided recipient's username, and text is set to the provided text. This message is then passed to the OTR send function. After the OTR send function returns the message is serialized and sent to the server to be forwarded along to the recipient.

One question that has not been answered in this section is why OTR send and receive functions are called on all of our messages? The goal of this application design was to separate the cryptographic operations from the functionality of the messaging as much as possible. This is hard to keep completely separate, for example the creation and publishing of the long term public keys is handled in the application. Yet, the messages are passed to the OTR send and receive to be modified. The OTR instance acts as middleware for the application handling all of the cryptographic operations and allowing the application to focus on sending and receiving messages. This design allowed for these two pieces to be developed in parallel and completely separate from each other. If you were to comment out these two function calls the application would function still be a fully functional chat application just without OTR. A lot of information is passed from our applications global state into the OTR send and receive functions so they can perform the operations they need to. The websocket, token, contact list, username, and long term public private key pair are all passed to these functions. What these functions do with all of this information and how these functions operate is covered in the following section.

3.3 OTR

The OTR implementation was written in Typescript and compiled targeting ES6 syntax. The only dependency the OTR implementation has is the JWCL wrapper that will be covered in the next section. This implementation consists of seven main stateless functions that implement each step in the OTR protocol and an OTR class that implements the OTR state machine for a conversation.

All seven of of these stateless functions have the same function signature that consists of three parameters local, network in, and network out. All functions return void, any errors are delivered by throwing the corresponding exception. Local is a data store that stays on the client, this is used for holding keys and other client side data. Local is not a data store for the state though, this is held in the OTR class that will be covered later in this section. Network in, is a read only data structure that is the input to the function coming from the network. Network out, is a passback parameter that is used to return the data that should go on the network after returning from this function. The decision to use separate parameters for input and output was because of the way javascript passes values. Javascript is a pass by reference language. Therefore modification of a variable passed in will be seen by that pointer in the calling function. The issue with that in this case is that the network variable must be kept clean by deleting the old data before adding new data to that object. It was much easier to instead treat the input as readonly and then instantiate a new object for the output that was then written too.

For the following sections describing the authenticated key exchange states the case that Bob is initiating an exchange with Alice and then sending her a message will be made. Bob and Alice are assumed to both have long term public keys called pubB and pubA that are published and available to each other. In this implementation that is handled by the application layer as mentioned in the previous section.

3.3.1 Authenticated Key Exchange State 1

In this state Bob starts by generating a random key that will be called R. Bob then generates an ecdh key pair called GX. The public part of GX is encrypted using R as that key, that value will be called aesGx. The public part of GX is also hashed using SHA256 that value will be called hashGx. Bob stores R and GX in his local storage. Bob also places GX in his key storage as his current key pair with Alice. Bob also generates his next key pair for the conversation and stores that as well. The network out object type field is set to ake1. The aesGx and hashGx fields of network out are set to the their corresponding values.

3.3.2 Authenticated Key Exchange State 2

In this state Alice starts by generating an ecdh key pair called GY. Alice stores GY in her local storage. She then stores the received aesGx and hashGx as well. Just like Bob, Alice places GY in her key storage as the current key pair and generates her next key pair. The network out object type field is set to ake2. The gy field of network out is set to the public part of GY.

3.3.3 Authenticated Key Exchange State 3

In this state Bob takes GY from the network in object. Bob uses GY and GX to derive random bits called S. Bob uses S to derive two encryption keys called C and C and four mac keys called M1, M1, M2. and M2. This process is completed by hashing S in various ways. Next the value MB is created. First an object consisting of four fields, gx, gy, pubB, and keyIdB is created. Gx is set to the public part of GX. Gy is set to the gy received from Alice. PubB is set to Bobs long term public key. KeyIdB is set to 1 as this is Bobs first key with Alice. This object is MACd with the

key M1 to create MB. Next the object XB is created which consists of three fields pubB, keyIdB, and a value called sigMb. SigMb is computed by signing MB using pubB. Next XB is encrypted using C to get the value called aesXb. Then aesXb is MACd using M2 to get the value called macAesXb. The six keys generated as well as Alices GY are placed in the local storage. The network out type field is set to ake3. The field R is set to the value of R generated during ake1. The fields aesXb and macAesXb are set to their corresponding values.

3.3.4 Authenticated Key Exchange State 4

In this state Alice takes R from network in. She uses this to decrypt aesGx sent to her by Bob in ake1. Alice then hashes GX and compares the hash to the one that was sent. If this comparison fails an exception is thrown with the message Error ake4: hashes do not match. If the comparison is a success Alice uses GX and GY to derive the same random bits for S as Bob did. She then hashes S in various ways to get the same six keys Bob computed. Alice first uses M2 to verify the MAC of aesXb. If this verification fails an exception is thrown with the message Error ake4: mac does not verify. If this verification is successful, XB is then decrypted using C. Alice reconstructs MB by signing the same values as Bob did with M1. Alice then takes pubB and uses it to verify the signature of mB. If this verification fails an exception is thrown with the message Error ake4: signature does not verify. If this verification succeeds Alice has now validated all of the information Bob sent in ake3. The next step is to send very similar object back to Bob. First MA is computed which is the MAC using M1 of gx, gy, pubA, and keyIdA. These are set to the same values as the were for Bob with the exception if pubA being Alices instead of Bobs long term public key. XA is again computed in a similar fashion as Bob computed XB. Alices creates the object with the values pubA, keyIdA, and the signature of MA using pubA. Alice encrypts XA using C1 to get aesXa. Alice then MACs aesXa using M2

to get macAesXa. Alice stores the keys computed from S and GX in her local storage. She also sets their key id field to the keyIdB Bob sent. She then stores GX in Bobs key storage. The network out type is set to ake4. The fields aesXa and macAesXa are set to their corresponding values.

3.3.5 Authenticated Key Exchange State 5

In this state Bob performs the verifications that Alice did in the first half of ake4. First Bob verifies the MAC of aesXa. If this verification fails an exception is thrown with the error message Error ake5: mac does not verify. If this verification succeeds XA is retrieved by decrypting aesXa. MA is then computed by using M1 to MAC the same values that Alice did which were gy, gx, pubA, and keyIdA. After this Bob uses pubA to verify the signature of MA. If this verification fails an exception is thrown with the error message Error ake5: signature does not verify. If this verification succeeds, Bob then sets their key id field to keyIdA and stores GY in Alices key storage. If all of these verifications succeed the authenticated key exchange is complete.

3.3.6 Exchange Data State 1

Now that the key exchange is complete Bob can send his message to Alice. Bob gets the plaintext from the message passed in by the application layer. Bob takes out Alices public key and his private key they agreed upon in authenticated key exchange, from his key storage. Bob then uses this public private key pair to derive random bits called S. He then hashes S in various ways to get four keys sendAesKey, recvAesKey, sendMacKey, and recvMacKey. Bob uses the sendAesKey to encrypt the plaintext to ciphertext. He then constructs and object called TA that has the send key id, the receive key id, the next dh key bob wants to use, and the ciphertext. Bob then uses the sendMacKey to sign TA getting the value macTA. Bob sets the type field to ed1

and sets TA and macTA to their corresponding values.

3.3.7 Exchange Data State 2

Alice has received Bobs message now and wants to see what it says. She starts by getting TA and macTA from the network in variable. Using the receive key id Alice pulls out her corresponding private dh key and using the send key id she gets Bobs public dh key from her key storage. Alice then uses this public private key pair to derive random bits called S. She then hashes S in various ways to get the same four keys as Bob sendAesKey, recvAesKey, sendMacKey, and recvMacKey except they are flipped. Alices receive is Bobs send. Alice uses the recvMacKey to verify macTA. If this verification fails an exception is thrown with the error message ERROR ed2: mac does not verify. If this verification is successful Alice uses the recvAesKey to decrypt the ciphertext and get the plaintext. She places this in her local storage for the application layer to display.

3.3.8 OTR Class

The OTR class is the go between that lets the application talk to the OTR functions. The class also has the responsibility of running the OTR state machine for each conversation.

The constructor takes no arguments and only creates an empty conversations object. The conversations object is a key value mapping between the username of who you are talking to and their corresponding conversation information. There are two functions that the OTR class has which are send and receive. They both take the same list of parameters which are ws, token, contacts, username, longKey, message. Ws is the websocket connection. Token is the JWT. Contacts is a list of all the active user objects in the system. Username is the username for the active user. LongKey

is public private key pair for the active user. Message is the message object being passed in from the application. The message object has the fields action, token, from, to, and text. The send and receive functions in this class add one field to the message object called otr, this holds all of the otr information from the ake1-5 and ed1-2 network out parameters. So how this is related to the function signatures of each of our states mentioned above is as follows. The conversations objects mapping is used as the local store, the otr object is the receiving case is the network in object, the otr object in the sending case is set to the network out object.

send

The send function has two cases to handle, which are a new conversation and a continuing conversation.

In the case of a new conversation, a mapping needs to be created in the conversations object between the recipient's username and relevant information. Key management information is created which is described in detail in the next section. The text of the message is buffered for use after the authenticated key exchange is completed. The otr variable is created with the type field being set to query. This is a request to start the authenticated key exchange.

In the case of a continuing conversation ed1 is called to encrypt the message. The the otr variable is set to the network out result of the function call.

A new message object is created with the fields action, token, from, to and otr. Action, token, from, and to are set to the identical fields from the message object that was passed in. Otr is set to the otr object that was created. This object is returned back to application to actually be sent.

receive

The receive function has six cases to handle, which are based on the type of message that is received. These are query, ake1, ake2, ake3, ake4, and ed1.

In the case of the type being query the otr variable is taken from the message coming in. This becomes the network in variable for the function call to ake1. The otr variable is now set to network out. This is where the websocket that was passed in to this function comes into play. The receive function actually sends a response on behalf of the user. This makes it transparent to the application that the key exchange is taking place. The response has the same action as the message, the token is set to the passed in token, from is set to the username, to is set to the from field in the passed in message since this is a response. Finally otr is set to the otr value that was saved after the completion of the ake1 function call.

In the case of the type being ake1, ake2, ake3, ake4 the same process is followed. The otr variable is pulled from the received message to be used as the network in value. The next step in the authenticated key exchange is called and the resulting network out is sent as the otr variable in the outgoing message.

For the cases of receiving ake3 and ake4 the message buffer that was created in the initial send is checked. If there is a message in the buffer this message is encrypted by calling ed1 and sent as well.

In the case of the type being ed1, ed2 is called on the received information to decrypt the message. The plaintext is added to the message data structure and passed up to the application for display to the user.

3.3.9 Key Management

Key management in the OTR implementation is handled by a few variables. They were reference above as key store. This section explains how this store operates and explains the process of cycling keys. The OTR classs representation of a conversation has six fields that relate to key management, these are ourLongKey, ourKeys, ourKeyId, theirLongKey theirKeys, and theirKeyId.

The ourLongKey and theirLongKey are the long term public private keys that are used during the key exchange.

The ourKeys, ourKeyId, theirKeys, and theirKeyId are much more interesting. This is the key store for our dh key pairs. Our keyId is initialized to be 2 since two keys were generated in either ake1 or ake2 depending on which side of the authenticated key exchange our side was. When ed1 is entered the key ourKeyId - 1 is pulled from ourKeys for use as the private key in the computation of S as described in the ed1 section. The key ourKeyId is pulled from ourKeys for use as the nextDh key. The receiving side is where key rotation takes place. There are two cases that cause a key roation. The first is if the recvKeyId is equal to ourKeyId then that means ourKeys has no new keys to send so one must be generated. First the oldest key is deleted, then a new dh key is generated and placed in ourKeys, and finally ourKeyId is incremented by one to reflect this process. The second state that causes key rotation is when sendKeyId equals theirKeyId, in this case the nextDh they send us is added to their keys and theirKeyId is incremeneted by one to reflect this change.

3.4 JWCL

JWCL was written in Typescript and compiled targeting ES6 syntax. JWCL is a wrapper that makes the native WebCrypto API easier to work with. JWCLs interface

was modeled after SJCL [1]. JWCL injects one top level variable, jwcl, into the global scope. JWCL uses base64 encoding heavily to ensure portability over a network. A result of this that raw JWCL output data is not usable by other cryptographic libraries without careful decoding. The jwcl variable has one top level function called random and four modules utils, hash, cipher, and ecc. Utils provides helper functions to translate between binary data, base64 encoded strings, and numbers. Hash provides access to sha1 and sha256 functions, and an HMAC class. Cipher provides access to an AES class. Ecc provides access to ECDH and ECDSA classes. JWCL uses classes when the cryptographic primitive needs to maintain state and functions when it does not. For example AES was implemented as a class so the key could be remembered between function calls but sha1 was implemented as a function because there is nothing to be remembered.

3.4.1 jwcl.utils

The utils module is focused on providing helper functions for type conversions between binary, string, and integer data types. These can be called from outside the library but their main use is internal. There are eight functions in utils, btos, stob, itob, btoi, btob64, b64tob, stob64, and b64tos.

Btos and stob convert between strings and binary data. Javascript now represents binary data in Typed Arrays. [16] Typed Arrays are not portable over a network so a string representation was needed. This was accomplished by using a Uint8 typed array. To go from binary to string an empty string was created. Then the array was looped through. At each byte String.fromCharCode was called to convert the numeric representation of the data to its corresponding character representation. [14] This worked well with the Uint8 type because the max value for an unsigned byte is 255 and that is well within the capabilities for fromCharCode to convert. This

converted character was then appended to the string. Once all members of the array had been converted the resulting string was returned. Stob works the same way just opposite. First an empty Uint8Array was created with a length equal to the string length. Each character in the string was looped through and string.charCodeAt was called to convert the character back to its numeric representation. [13] Once all characters have been placed in the array, the array is returned.

Itob and btoi convert between binary data and integers. At the time of this implementation there was no way to do math with Typed Arrays. These functions support up to 32 bit unsigned integers. Itob works by creating a Uint8Array to hold the result. Each of the four bytes is then filled in by taking the floor of the division of the number and 2 to the power of bits were at and then repeating with the remainder. For example 259 would be divided by $2^2$4 first resulting in 0 so that would go in the high byte. Then $2^1$6, again 0 in the next byte. Next 2^8 which would result in 1 remainder 3 so one would go in the second to last byte. Finally 3 is divided by 2^0 so 3 goes in the low byte position. This array is then returned. Btoi works in reverse. First a total is initialized to 0. Each byte in the array is multiplied by its respective power of two and then added to to the total. This total number is then returned. This is really just simple binary math to get these values converted.

Btob64, b64tob, stob64, and b64tos all use the native window.atob and window.btoa functions. [12] These functions provided base64 encoding. This is used to ensure url/network safe strings for passing cryptographic data over a network connection. Btob64 first converts the binary data to a string using btos, after this is done the result is passed into window.btoa to get the base64 encoded string. B64tob works in the opposite way, the base64 string is decoded using window.atob then converted to binary using stob. B64tos and stob64 directly call the window functions window.atob and window.btoa respectively. These were implemented simply to have functions complete this action that matched the naming scheme of the rest of the

utility functions.

The utils module is fully responsible for all data type conversions and the rest of the JWCL library makes heavy use of its features.

3.4.2 jwcl.random()

The random function provides generation of cryptographically secure random numbers. The function takes one parameter which is the number of bytes to generate. A Unit8Array of that length is generated. The WebCrpyto API random function is called with the array as a parameter. The resulting array is encoded using the bto64 function and the resulting base64 encoded string is returned. Because of this encoding scheme these random numbers must be decoded before use in other contexts. The library handles this for any use of random numbers in the APIs.

3.4.3 jwcl.hash

The hash module consists of two static functions, sha1 and sha256, and one class, hmac.

Sha1 and sha256 both take a plaintext string as input and return a Promise of type string as output which is the digest. The output string is computed using the WebCrypto digest function. This function takes a name parameter as a string and a plaintext parameter as a TypedArray. The strings SHA-1 and SHA-256 are passed in to represent their respective hash functions. The stob is called to encode the plaintext string into its related TypedArray. The output of the hash function is again a TypedArray so translation to a string is needed. This is done using the btob64 function.

The hmac class has a constructor that takes the key as string as input. This key

should have been generated using the random() function. This key is stored in a local variable for use throughout the lifetime of the class. The hmac class has two functions, sign and verify.

Sign returns a Promise of type string which is the signature. Sign takes one parameter which is the plaintext string to be signed. First the base64 encoded key needs to be decoded and translated to a TypedArray. This is done by calling the b64tob function. Next a WebCrypto CryptoKey needs to be created. The import key function is used to translate the TypedArray representation of our key to a CryptoKey representation. The import key takes four parameters, name, hash, extractable, and usages. The name is set to HMAC since that's what this key will be used for. The hash is set to SHA-256 so that hash function is used in the HMAC. Extractable is set to true. Usages is set to sign since that's the action that is being performed in this function. After this is done the WebCrypto sign function is called. This function takes three parameters, name, key, plaintext. The name is set to HMAC since that is the signature algorithm this class performs. The key is set to the CryptoKey. The plaintext is set to the Typed Array of the plaintext string that was passed in which is generated by calling the stob function. The result of the WebCrypto sign is a TypedArray. This is translated to its base64 string representation using btob64 which results in the signature.

Verify returns a Promise of type boolean which is whether or not the signature verifies. Verify takes two parameters which are the signature and the plaintext both of type string. The process of creating the CryptoKey is exactly the same as in sign except the action parameter is set to verify and not sign. After this is done the WebCrypto verify function is called. This function takes four parameters name, key, signature, and plaintext. The name is set to HMAC since that is the verification the class performs. The key is set to the CryptoKey. The plaintext and signature both came is as strings and this function takes Typed Arrays. Since the resulting

signature in sign was base64 encoded b64tob is called to translate the signature to a Typed Array. The plaintext on the other hand is a regular string so stob is used. The result of the WebCrypto verify is a boolean and that is what is returned.

3.4.4 jwcl.cipher

The cipher module consists one class, aes.

The aes class constructor takes the key as a string as input. The key should have been generated using the random function. During construction a private internal counter is initialized to zero. Aes also has two private internal constants, counter bytes, and block size bytes. Counter bytes is used to determine how many bytes the counter will be. Block size bytes is used to compute the next counter, a process that will be covered later in this section. They are both set to 16 bytes. As long as the underlying AES implementation does not change block size bytes should always be 16. The aes class has two functions, encrypt and decrypt.

The encrypt function takes the plaintext string as input and returns a Promise of type string. The WebCrypto API needs Typed Array input for both the string and counter. These are created by calling stob and itob on the plaintext and counter respectively. Just like in the hmac class the key must be imported into type CryptoKey. The parameters that differ in this case are the name and usages parameters. For aes the parameter is set to AES-CTR to use the AES algorithm using counter mode. The usages is set to encrypt since that is the action this function performs. Once this is done the WebCrypto encrypt function is called. This function takes 5 parameters, name, counter, length, key, and plaintext. The name is set to AES-CTR because that is the algorithm chosen for this implementation. The counter is set to the counter that was translated to a TypedArray. The length is set to 128, this is the number of bits in the counter. The key is set to the created CryptoKey. The plaintext is set to

the translated TypedArray of the input plaintext. The output of the encryption is another TypedArray that holds the ciphertext. To create the output the next step is to append the counter to the ciphertext as the first block. The counter is then update using the computation of, counter equals counter plus the ceiling of the length of the message in bytes divided by bytes per block. Finally the ciphertext is translated to a bas64 encoded string using btob64 and returned.

The decrypt function takes the ciphertext base64 encoded string as input and returns a Promise of type string. The first step is to translate the base64 encoded string back to a Typed Array using b64tob. Next the counter is separated by removing the first block of the ciphertext. The CryptoKey is created the same way, with the exception of usages being decrypt because that is the action this function performs. The WebCrypto decrypt function is called which takes the same five parameters as encrypt except the plaintext is now ciphertext. The values of these are all the same as encrypt except the plaintext Typed Array is replaced with the ciphertext Typed Array. The counter is then updated to the max of the internal counter and the received counter. This is done to keep an instance of aes in sync with another instance across a network. The decrypt function returns a Typed Array for the plaintext so that is translated to a plain string by calling btos and returned.

3.4.5 jwcl.ecc

The ecc module consists of two classes, ecdh and ecdsa.

The ecdh constructor takes one parameter which is a key of type eccKey. This type is an object with two fields, public key and private key both of which are strings. Ecdh has one static function generate and one function derive.

The static generate function is used to generate ecdh key pairs. This function was implemented statically so that key generation could be done separately from object

construction. This is necessary because key generation is an asynchronous process and object construction is not. This function takes no parameters and returns a Promise of type eccKey. Since the key generation process for ecdh keys is not just generate random numbers and use them, this function is provided. The WebCrypto generateKey function is called. This function takes four parameters name, namedCurve, extractable, and usages. The name is set to ECDH. The named curve is set to P-256. This curve is approved for use by NIST for the federal government. [18] Extractable is set to true. Usages is set to deriveBits so that our key pair can be used to generated random bits. As mentioned in the constructor the eccKey type holds the public private key pair as strings. The process of translating the CryptoKey pair to string is the same for both public and private key. First they key is exported under the Json Web Key format. This format is serialized using JSON.stringify. The string output is then base64 encoded using stob64. This gives a network safe representation of the keys. The public and private key strings are then placed in an eccKey object and returned.

The derive function takes one parameter which is a base64 encoded public key string. This value should come from the public part of the eccKey that was created using generate. This function returns a Promise of type string which is the derive bits. These can be used as a key in the aes and hmac classes or any other place random bits are needed. The process done in key generation first needs to be reversed so the CryptoKey representations of the public and private keys are available. This is again the same for both public and private keys and is done by calling the WebCrypto importKey function. This function takes six parameters, type, key, name, namedCurve, extractable, and usages. The type is set to Json Web Key by the string jwk. Key is the JSON representation of the key. This is gotten by calling b64tos and then JSON.parse on the result. The name, namedCurve, extractable, and usages are the same as in generate with one important difference. When importing the public

key usages is left blank. There is nothing in the WebCrpyto API spec that says why this is but the code does not function if this value if set to derive. Once the keys are imported the WebCrypto deriveBits function is called. This function takes three parameters, name, publicKey, and privateKey. Name is set to ECDH, the public and private Keys are set to their corresponding CryptoKeys. This function returns the derived bits as a TypedArray so they are translated to a base64 encoded string using btob64. The resulting string is returned.

The ecdsa constructor takes one parameter which is of type eccKey, exactly the same as ecdh. Ecdsa has one static function generate and two functions sign and verify.

The static generate function takes no parameters and returns a Promise of type eccKey. This function works identically to that of ecdh with the only exception being the name is set to ECDSA during the call to the WebCrypto generate key function.

The sign function takes one parameter of type string as input. This is the plaintext to be signed. This function returns a Promise of type string which is the signature. The first step of sign is to decode the private key from the base64 string to the CryptoKey representation. This is done the same way as in ecdh except name is set to ECDSA. The WebCrypto sign is then called. This is the same function that hamc uses. This function takes four parameters, name, hash, private key and plaintext. The name is set to ECDSA. The hash is set to sha256. The key is set to the imported private key. The plaintext is set to the input plaintext translated to a Typed Array. This is done by calling stob on the input parameter. The resulting signature is also a Typed Array so btob64 is called to get the base64 encoded string result. This is returned.

The verify function takes two parameters signature and plaintext and returns a Promise of typed boolean signifying if the signature verified or not. The public key is

used for signature verification so that is converted to a CryptoKey in the same way the private key was in sign. Next the WebCrypto verify function is called. This function takes five parameters, name, hash, public key, signature, and plaintext. The name and hash are set to the same values as sign. The public key is set to the CryptoKey representation of the public key. Both the signature and plaintext expected Typed Array so the input must be converted. This is done by calling b64tob for the signature since it was base64 encoded, stob is used for the plaintext. The boolean result of verify is returned.

Chapter 4

VALIDATION

The application was tested both manually and with automated tests. Both types of testing were used to get as much test coverage as possible.

4.1 Automated Tests

The automated tests were built using the QUnit Javascript testing framework and ran in the browser.[9] They can be accessed by navigating the browser to /test, after this all tests will run automatically and give the user access to the QUnit test report user interface. The automated tests were primarily unit tests. They were used as regression tests as well during development of the application.

4.1.1 Server Tests

The server tests, test the /api/signup and /api/login api endpoints.

/api/signup

The first set of tests done against the server are to signup three users, Alice, Bob, and Charlie. This is done by sending three asynchronous requests to the server. The URL for all of these are /api/signup, the method is POST, and the body is a JSON string containing the username and password combination of the user to be created. After these requests come back the status codes as checked to be not equal to 500. The reason the status code is not checked to be equal to 200 is so that these tests can be ran multiple times without restarting the server. After the first signup of a user the server responds with status code 400, so if the status code was checked to

be equal to 200 the tests would fail every time after one run.

/api/login

Once the users have been signed up there are tests to check if they can now log in. This is done by sending asynchronous requests to the server again. The URL for these is /api/signup, the method is POST, and the body is the same as sign up, with the JSON string containing the username and password. After these requests come back the status codes are checked to be equal to 200. There is no issue with users logging in multiple times so the multiple run issues in sign up do not arise in the login tests.

Because of the asynchronous nature of AJAX requests promises are used here to ensure ordering of signup before login every time.

4.1.2 Application Tests

The application tests are very short. They check for the existence of the global data store, the chat page implementation, and the login page implementation. The rest of the application is tested manually and will be discussed in that section.

4.1.3 OTR Tests

The OTR tests work at two different levels. The first set of tests validates the implementation, these tests call the ake1-5 and ed1-2 functions directly. The second set of tests validates the api, these test only call send and receive.

Implementation Tests

The implementation tests by setting up all of the state information. This test code actually heavily influenced the development of the send and receive functions as these

tests were written prior to those functions being implemented. To set up state information Alice and Bob have object setup for them respectively. A network in and network out object are created to spoof network communication. Key management objects are created for both Alice and Bob. Finally two ECDSA keys are created as the long term public keys for Alice and Bob.

After the setup is complete the tests go through the authenticated key exchange, by separately calling ake1-5 alternating between Alice and Bob respectively. Once the key exchange a message is sent from Alice to Bob. This message is this is a message. After Bob receives the message by calling ed2 it is compared to the original message for equality. Bob the sends a response message this is a response and that is checked by Alice.

API Tests

The API tests work very similarly to the implementation tests with the exception of only send a receive being called. The setup for the API tests was to create a fake contacts list that had Alice and Bob in it as well as their public keys. For these tests a spoofed websocket implementation was created. This implementation has the same send and receive api. The difference is that it doesnt go over the wire, it just buffers the message and returns it on the next call of receive. This was an easy way to spoof the websocket so the implementation code didnt have to be modified for testing.

The API tests run through the same scenario as the implementation tests where the authenticated key exchange is performed and then Alice sends Bob a message and Bob sends Alice a response. At every step the type and to from parameters are checked. Once the messages are passed those are also checked to make sure the correct message is received.

4.1.4 JWCL Tests

JWCL has tests for all of the modules. The interesting part of testing JWCL is the tests were not meant to test the underlying cryptographic primitives. This job is left up to the WebCrypto API implementers, and in this project's case that is the Chromium team. JWCL was instead tested for functionality.

The random function was tested by generating a random 16 byte value. The result is then checked to make sure it is of type string and length equal to ceil(16/3)*4. This is the equation for the length of a base64 encoding. Since the random function returns a base64 encoded random string that is the expected length.

The hash functions sha1 and sha256 were tested by hashing the string abc. This value was compared to the base64 encoded string from online hash function tests vectors. [19]

The HMAC class was tested by attempting to sign and verify the text important message. First a random 16 byte value was generated as the key. An HMAC instance was constructed with the key. The instance of HMAC is used to sign the key. Once the signature is returned the verify function is called twice. First with the string important message to make sure that the signature verifies as true. Next with the string important message changed to make sure the signature fails to verify in that case.

The AES class was tested by attempting to encrypt and decrypt the test secret message. A 16 byte key was generated and then used to construct and AES class instance. The plaintext was then encrypted. The ciphertext was then decrypted and check for equality against the original string secret message. This is what was meant in the introduction to the JWCL tests by performing functional tests. There is no test vector used in the test to ensure that the key and plaintext pair resulted in the

correct ciphertext. As mentioned earlier that testing is left up to the WebCrypto implementation tests. JWCL only cares that the AES class is able to encrypt and decrypt that's it.

The ECDH class was tested by generating random bits. Two ECDH keys were generated. The first key was used to created an instance of ECDH. The derive function was then called with the public part of the second key. The resulting generated bits then had the same tests as random applied to them. The length and type were checked.

The ECDSA class was tested in the same fashion as the HMAC class with the exception of two keys being used. One to sign and the other to verify. Again the string important message was used. Verifying both important message for correctness and important message changed for expected failure was also performed.

4.2 Manual Tests

The manual tests were used to test the end to end functionality of the application. They were also used to stress test the app more than the automated tests. The manual tests started by opening up a browser navigating to the app and logging in as Alice. After this an incognito window was opened. This sets the application in a different context so that Bob can log in on that window. There are three scenarios that were commonly run.

4.2.1 Scenario 1

The first scenario was the most basic. In this scenario the app was used as a chat application commonly is and messages were sent back and forth. In this test no logs were examined. This test was meant to test the application from the user's

perspective and make sure nothing was amiss to the user. Some of the values looked at in the test were the To and From values were always correct. Another was that the message text was exactly what was sent.

4.2.2 Scenario 2

In this test the main focus was the key cycling. In this test all messages are send as the same plaintext test message. Alice starts by sending two of these messages in a row to Bob. Bob responds with two in a row. Then Alice sends one more back to Bob. The logs are then pulled up and examined on the server implementation. What is looked at is the ciphertext being sent back and forth. First none of the ciphertext for the five messages should be the same even though the plaintext was the same. Second the counter should have incremented for the second message for both Alice and Bob. Third when Alice responds the counter should have reset to zero but the ciphertext should still be different than her first message under the counter zero because now the encryption key should have been cycled. If all of these pass its good.

4.2.3 Scenario 3

In the third scenario Charlie is logged in. Alice, Bob, and Charlie now all talk to each other. This test is used to make sure that users are able to have multiple conversations at the same time. The To From and message text values are check here to make sure there is no routing or message mix up between conversations.

Chapter 5

FUTURE WORK

5.1 Server

The server part of this application works well but is missing a lot of features that a deployable server side would need. These features are a persistent data store, HTTPS support, and separation of services.

5.1.1 Persistent Data Store

The server uses a python dictionary object as a database right now. The result of this is that every time the server is restarted the database is reset. If the server were to actually be deployed as a real application this would not work. The data model is simple, there is just storage of username, password hashes, and public keys. This could easily be built into a relational or noSQL database engine. One note here is that the websocket instance would not be stored in the persistent storage. This is because websocket instances are not serializable and as soon as someone logs off the instance is destroyed.

5.1.2 HTTPS Support

HTTPS support is a critical part of the security of this application. The OTR protocol relies on the authenticated transfer of the long term public keys. Since the server handles this action HTTPS between the server and the clients is important to ensure the public keys are correct and OTR can complete the authenticated key exchange correctly. HTTPS support could be added by placing the Python application behind a reverse proxy such as Nginx. [17] Nginx has HTTPS support out of the box and is

easily configurable to use that. This is how HTTPS would be added.

5.1.3 Service Oriented Architecture

Right now both the authentication and messaging take place in the same application context. While there is nothing inherently wrong with this, separating them has benefits. The separation of responsibility between two applications would make development changes easier to reason about.

The changes needed to implement the server as a service oriented architecture start at the database and go from there. First the database would have to be separated. There would need to be two databases, one holding the usernames and password hashes and another holding the public keys and any other messaging transactional data. Next the Json Web Token implementation would have to be update to support public private key pairs. Separating the message service from the authentication service would entail the messaging service gets the authentication public key and uses that to verify JWT signature. Other than this is would be just writing two separate Python files and running them on different ports. Then using Nginx as the proxy the correct service would be ran based on the URI.

5.2 Application

The application part of the this implementation is very barebones and was built to show the functionality of the underlying OTR and JWCL implementations.

The first improvement would be to add an interface for users to interact with signup API on the entry page. Right now test users are signed up through the tests scripts making calls to the API. This would not be a complicated undertaking but is an improvement to make the application usable to real users.

On the chat page, the app has a list of the available users and their status at the top, this information is fine but the way it is displayed could be better. An issue can arise when the user tries to send a message. Right now the user must correctly type in the desired recipient's username. If there is a typo by the user their message would not reach the desired recipient and they could become frustrated with the application. A feature where the user can click on the desired recipient or was provided with a searchable dropdown in the To form field would help alleviate this issue.

Another improvement in the chat UI would be the concept of conversations. The chat box now just appends the most recent message with the To and From users next to the plaintext. It would make it much more pleasant to view conversations if they only consisted of one person. Think back to your old flip phone and how annoying texting was before conversations were added to phones.

The application could also fail more gracefully. Right now an error message is displayed to the user but sometimes and error can leave the application in a bad state. In these cases a page refresh is needed to reset the client. Error handling needs improvement to ensure that the application is always left in a functional state.

There are message storage features that the application could provide but these have security considerations so they are discussed here and left up to the future worker to decided if they should be implemented. The feature discussed here will be simply storing old messages. OTR discards all old key material so as soon as a message is read and the keys are cycled, that message can never be retrieved again. From a user perspective this could provide a poor experience if every time the application was closed all past message history was deleted. A option to alleviate this problem could be to use a KDF on the user's password and store the messages on the client by encrypting them using the symmetric key generated by the KDF. This is a reasonable option, but one thing to consider is what do you do in the case of the user forgetting

a password? Now all of the saved messages are still lost. This shows how storing anything in this application leads to a lot of decisions about the tradeoffs of security and usability.

5.3 OTR

5.3.1 Differences from OTR Protocol

This implementation diverges slightly from the protocol description. [23] There are two areas where the implementation differs. The first is the use of elliptic curve cryptography for the diffie hellman and dsa primitives. The second is the state machine handling the conversation.

In the protocol the authors use diffie hellman and dsa over the set of integers. This implementation used the elliptic curve set, simply because chromium has not and will not support either of these algorithms over the set of integers in the future as according to the developer documentation is No longer part of the spec. [22] A side effect of this is that public private key pair generation in ake1 and ake2 is done by calling a generate function rather than doing the multiplication of g to the x and y respectively ourselves. The results are the same though as the public private key pair are still available to the implementation.

The state machine was implemented in a simpler manner because of the limitations of this implementation. In the protocol specification, the state machine is more fault tolerant and more generic. This implementation is more constrained in what input it can handle. An example can be used to illustrate this. In this implementation if ake2 is received but ake3 was expected the implementation would crash. The protocol would handle this situation based upon its internal state variables. An interesting thing about the protocol is that there are situations in which ignoring the message is

the specified action. The reason behind this implementation not currently handling this is the fact there is not API. Since users interact through the application layer with OTR in order messages are guaranteed.

The state machine needs to be improved to function to the specification of the protocol if the OTR implementation is to be completely decoupled from the application implementation. This work would make the implementation more robust and specification compliant as well as more portable for others to use in their applications.

5.3.2 Socialist Millionaires Protocol

The OTR specification also has another protocol in it called the Socialist Millionaires Protocol. This protocol is used to test if x equals y without revealing any other information other than the value of x equals y. So if x and y are secret and if x is not equal to y, Alice and Bob gain no information about the others secret. This can be used during OTR if Alice or Bob suspect impersonation or man in the middle attacks. They can use the Socialist Millionaires Protocol to check secret information for equality. There was no work done on this protocol in this thesis but it could be a nice future project to implement.

5.4 JWCL

JWCL was developed to make the native WebCrypto API easier to work with. The main issues that JWCL tries to solve from working with the WebCrypto API directly are transparent handling of binary data, secure defaults for all optional parameters, and provide a class based interface.

5.4.1 Binary Data

The JWCL implementation in this project is a start here. The binary data is handled transparently in all the functions but it is base64 encoded. To get back to the Typed Arrays of Javascript calls to the utility b64tob functions must be made. In the future the output type of JWCL would be customizable. In the case of sending output over the network base64 encoding is a nice approach for its url safety. In the case of working only in one browser it is a lot of overhead to constantly encode and decode base64 to work on the raw data. In this case it would be nice to be able to configure JWCL to output the raw binary data.

The JWCL implementation handling of keys, especially public and private keys faces a similar issue. It is a non configurable option that what is returned is a base64 encoded representation of the keys. This is fine if you are only working with JWCL. It is not fine in the case that the raw API would like to be mixed with JWCL though because the decoded process to get back to a CryptoKey instance is non trivial. In the future this would be another customizable parameter. This parameter would allow the user to configure what output format she expects her key output to be.

5.4.2 Secure Defaults

The JWCL implementation has a lot of default parameters. An example is in the HMAC class sha256 is used as the hash. This is non configurable. While this is a sensible default, what is someone wanted to use sha1 for compatibility with other systems? The WebCrypto API is highly configurable offering a parameter for a lot of different options. In the future JWCL will continue to have secure and sensible defaults but allow for user customization of all the parameters to the WebCrypto API. This strategy will attempt to protect the novice cryptographer from himself while allowing the expert cryptographer full access to the underlying API.

5.4.3 Class Based Interface

The JWCL implementation aims to use classes whenever there is stored keys. This allows the user to put the key in the constructor and then forget about it. The WebCrypto API is completely stateless so the user would have to remember to use the proper key for every call. This is pretty stable for all the chosen classes that were implemented.

JWCL also needs more functions and classes. Right now there is support for sha1, sha256, HMAC with sha256, AES-CTR, ECDH with curve p-256, and ECDSA-with curve-p256. Adding classes or at least configurable parameters to allow access to the full WebCrypto API is needed. The largest change would take place in the AES class. Because the CTR implementation uses a stateful counter, the counter must be managed. In the future an extended implementation to, for example AES-CBC would now use an IV and would be completely stateless. To keep the user friendly interface intact the difference between these two cases should be transparent to the user of JWCL. This will be an interesting challenge to work on.

JWCL is a project I intended to separate from this implementation and continue working on. I plan to make the changes discussed in this section as well as improve upon the test suite in the validation section and then open souce JWCL for others to use.

5.4.4 Typescript Issues

Typescript had a issue with the typing for the Webcrypto API. The main issue was the types for EcKeyGenParams in the ecc generate function calls. The Typescript typings file had the name of the parameter as typedCurve when the correct name was supposed to be namedCurve. This lead to the issue of either clean compilation

and broken code, or broken compilation and clean code. The second was what this project went with for now but in the future this problem should be solve. So I fixed it with this pull request to the Typescript typings github. [32]

Chapter 6

CONCLUSION

BIBLIOGRAPHY

[1] Stanford javascript crypto library, 2009. [Online; accessed October-2016].

[2] N. Bevacqua. Understanding javascripts async await, 2016. [Online; accessed March-2016].

[3] N. Borisov. Off-the-record communication, or, why not to use pgp. Technical report, University of California Berkeley, 2004.

[4] djafygi. Webcrypto examples, 2015. [Online; accessed March-2016].

[5] I. Fette. The websocket protocol. 2011.

[6] Google. Google allo, 2016. [Online; accessed October-2016].

[7] P. Hoffman and W. Wijngaards. Elliptic curve digital signature algorithm (dsa) for dnssec. Technical report, 2012.

[8] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). Technical report, 2015.

[9] T. jQuery Foundation. Qunit: A javascript unit testing framework., 2016. [Online; accessed September-2016].

[10] B. Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[11] D. C. Litzenberger. Pycrypto-the python cryptography toolkit. *URL: https://www. dlitz. net/software/pycrypto*, 2015.

[12] MDN. Base64 encoding and decoding, 2016. [Online; accessed June-2016].

[13] MDN. String.prototype.charcodeat(), 2016. [Online; accessed June-2016].

[14] MDN. String.prototype.fromcharcode(), 2016. [Online; accessed June-2016].

[15] Microsoft. Typescript - javascript that scales, 2012. [Online; accessed June-2016].

[16] M. D. Network. Javascript typed arrays, 2015. [Online; accessed March-2016].

[17] NGINX. Nginx webserver, 2016. [Online; accessed October-2016].

[18] NIST. Recommended elliptic curves for federal government use, 1999. [Online; accessed September-2016].

[19] NIST. Secure hash algorithm, 2016. [Online; accessed September-2016].

[20] OWASP. Using rfc2898derivebytes for pbkdf2, 2015. [Online; accessed September-2016].

[21] OWASP. Authentication cheat sheet, 2016. [Online; accessed May-2016].

[22] T. C. Projects. Webcrypto, 2016. [Online; accessed March-2016].

[23] C. Punks. Off-the-record messaging protocol version 3, 2004. [Online; accessed September-2016].

[24] K. Reitz. Elegant websockets for your flask apps., 2010. [Online; accessed March-2016].

[25] A. Ronacher. Flask web development one drop at a time, 2010. [Online; accessed January-2016].

[26] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 373–381. IEEE, 2009.

[27] O. W. Systems. Advanced cryptographic ratcheting, 2013. [Online; accessed March-2016].

[28] O. W. Systems. Open whisper systems partners with google on end-to-end encryption for allo, 2016. [Online; accessed October-2016].

[29] trevp. Double ratchet algorithm, 2016. [Online; accessed March-2016].

[30] w3ci. Web cryptography api, 2016. [Online; accessed June-2016].

[31] Wikipedia. Latex — wikipedia, the free encyclopedia, 2011. [Online; accessed 13-February-2011].

[32] J. Wilson. changed eckeygenparams from typedcurve to namedcurve to fix issue 135, 2016. [Online; accessed July-2016].