Final Report

# Twoefay

*A Multi-factor Authentication System for Aerospace Corporation*

*Chris Orcutt: 204 305 633*
*Jonathan Woong: 804 205 763*
*Rossen Chemelekov: 504 216 616*
*Mengfei Wu: 104 152 503*
*Vu Le: 004 497 690*
*Anthony Nguyen: 703 877 763*

University of California, Los Angeles
CS130: Software Engineering, Spring 2016
Professor Paul Eggert
Client: Mustafa Al-Ammar

Feel free to use the IEEE format or any format you like. Also, you're welcome to include sections from your SRS and Design doc, but make sure that they logically flow (i.e. don't just copy paste).
Single-spaced, Times New Roman, 12 pt.

## Introduction:

- explain the problem,
- existing solutions,
- summary of your solution

The Internet provides access to many resources and services which, while convenient, are still vastly insecure, relying primarily on a username and password to permit access. Although better than nothing, this is hardly secure as new and powerful tools are built and perfected each day which can sniff out, crack, and even bypass password-protected sites.

Most Internet users today don't have technical background and are unaware of the potential threats, so they use passwords that are simple to remember, rather than difficult to crack, and even reuse passwords across multiple sites, which leaves them vulnerable to malicious attacks by hackers. This is why passwords are not sufficient security for services that contain sensitive user information, such as credit card information, social security numbers, medical history, etc.

Numerous tech giants have integrated their own two-factor authentication solutions in their services, which are easy to setup and use. For example, users of Google, Facebook, and most banks can enable two-factor authentication in their settings where upon each login they must type in a 6-digit code that the service sends them via text message or email. There are even independent companies, such as Duo Security, that allow users to set up an account and enable the sites they wish to have two-factor authentication on. While versatile and secure, these services are not always trivial to set up and may be intimidating for the average Internet user.

This is where Twoefay comes in to provide vastly improved multi-factor security, while adding little complexity to the authentication process. The Twoefay service aims to provide security without sacrificing convenience, its innerworkings are therefore mostly invisible to the end-user. A user of a particular website, integrated with our service, who wishes to take advantage of the extra security must simply install the Twoefay iPhone app using the token provided upon registration at the site and the system will automatically set up the service. Then, when the user attempts to login from that point forward, they will receive a notification on their iPhone, use their fingerprint to authenticate, and access to the website will be granted.

## Software description:

(BRIEF 4 pages max)

- functional reqs,
- non functional requirements,
- use cases,
- architecture diagram and explanation,
- important components design with activity/sequence diagram

**Functional requirements:**

- The client website can authenticate the user by having the user press the touch ID on their iPhone, which is then verified with the dev_token in the database.
- The Twoefay mobile application must be able to communicate with the APN server to obtain the dev_token and communicate with the Twoefay server for token verification.
- The client website must be able to communicate with the Twoefay server to authenticate the user who is trying to log in.

**Non-functional requirements:**
- All the communications must be using HTTP/2 and TLSv1.2 for secure data transmissions.
- Data stored on the database of the Twoefay server, such as usernames, id tokens, dev tokens, must be safe and consistent with the users' accounts and their iPhones.
- The Twoefay server should respond to HTTP requests within a reasonable amount of time.

**Use Cases:**
1. Client
    a. A web administrator can use the Twoefay system to strengthen the security of a website which has an existing log-in system for users. The web administrator will be required to integrate the system by making HTTP requests to the Twoefay server at certain stages in the account creation and log-in process. These include when a Customer of the website signs up for an account and when a Customer attempts to log-in.
    b. An enterprise can integrate Twoefay with their existing system account log-in process which may not necessarily be web-based. The HTTP requests needed are the same as for the web administrator.
2. Customer
    a. A Customer of a website can sign up for an account and download the Twoefay iOS app which he/she will register using an ID token provided by the website. Upon a successful username-password match log-in attempt to the website, the Customer will receive an Apple Push Notification from the registered app on which the Customer can then use the fingerprint reader to authenticate.
    b. A Customer of an enterprise system will perform the same action flow as the Customer of a website but for the enterprise system log-in.

**Architecture**

The Twoefay authentication system consists of 3 major subsystems: an orchestration server, a push notification server, and an iOS client. The orchestration server is the main point of contact for all pieces of the system. It maintains a mySQL database that stores user information including username, email, phone number, user tokens for internal use, and device tokens for apple push notifications.

The server exposes the following endpoints:

| **Endpoint** | **Action** |
|---|---|
| /register | The register endpoint is used to register a new username in the database. |
| /verify | This endpoint is used to obtain the device token from the iOS application, which can then be used to send notification to the user’s phone via APN calls. |
| /login | The verify endpoint is used to initiate authentication. If the user has a device token registered, the user receives a TouchID authentication request, else the user receives a one time password via SMS (or email). |
| /auth | The auth endpoint is used to determine when a user has successfully authenticated with TouchID. The requesting entity continuously polls the endpoint until successful authentication or the authentication request expires. |
| /backup | The backup endpoint is used when authentication is carried out via one time password. The user will pass the one time password to the requesting entity which then uses the backup endpoint to verify that the issued and received one time passwords match. |
| /success | The success endpoint is used to indicate that the user has successfully authenticated. |
| /failure | The failure endpoint is used to indicate that the user has failed to authenticate. |

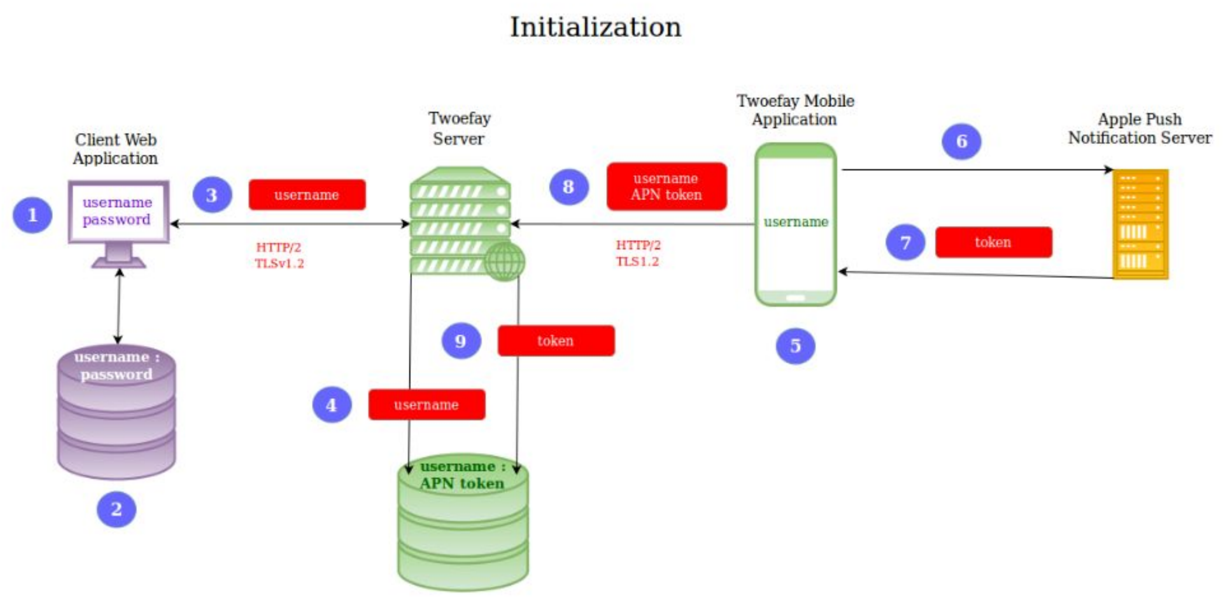
The push notification server is a client/server based microservice that handles delivery of push notification requests to the Apple Push Notification service. The server accepts incoming requests using HTTP/1.1 and sends outgoing requests to the Apple Push Notification servers using HTTP/2 over TLSv1.2. The server exposes one endpoint (the root endpoint) and expects POST requests with JSON encoded data specifying the alert and recipient of the push notification. The server was written in Python, uses the Flask web framework for handling requests and responses, and uses the nghttp2 library for all HTTP/2 communication.

The iOS application allows for biometric authentication via the TouchID system. Additionally, it serves as a medium to distribute one time passwords (via SMS) should the user reject push notifications or if the Apple Push Notification service is unavailable. The Twoefay system uses notifications to signify authentication requests from the orchestration server. Upon receipt of the notification, the user presumably opens the application and interfaces with the TouchID fingerprint scanning system to authenticate. The application sends the results of the authentication attempt to the orchestration server using HTTP/2 over TLSv1.2.

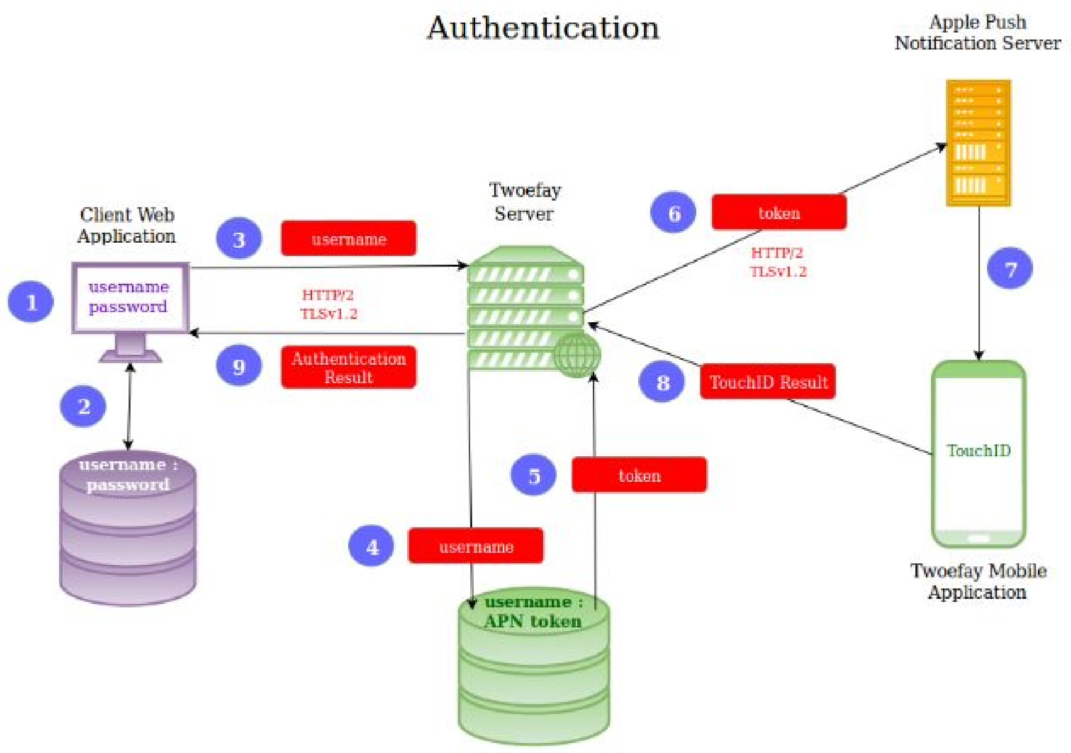
Although not officially part of the Twoefay system, it should be noted that to test our system, we created a fully function client website that illustrates how the Twoefay system could be used in production. The website interfaces directly with the orchestration server using HTTP/2. It was written in Python using Django.

**Primary Flows**

The Twoefay system contains two primary flows: initialization and authentication. Initialization occurs when a new user is signing up for multifactor authentication. Authentication occurs when a user wants to access restricted content.

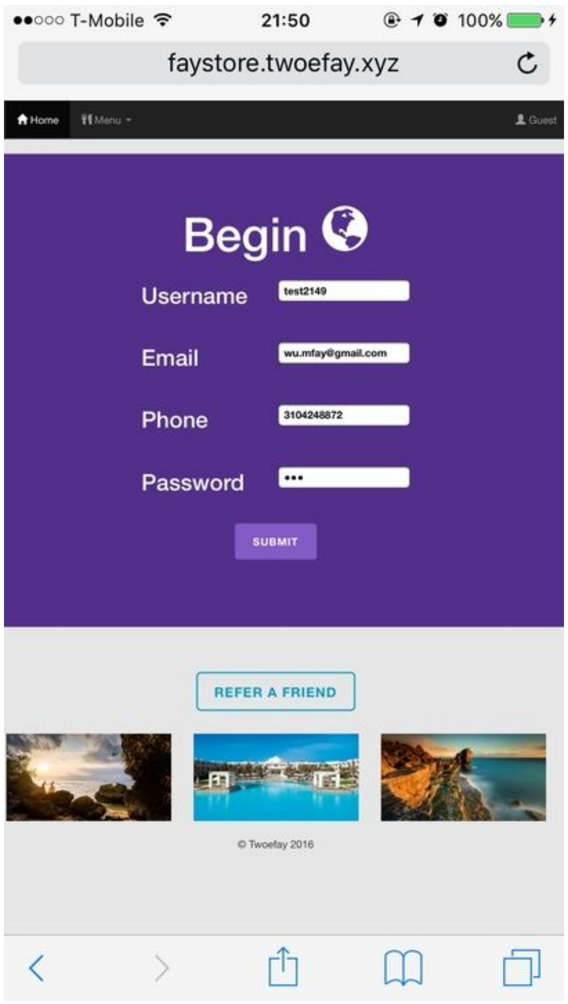


1. User signs up for website using a username and password
2. Website saves username and password in database maintained by website developer
3. Website informs Twoefay server of new username
4. Twoefay server adds username into database maintained by Twoefay
5. User signs into iOS application using the same username
6. iOS application requests APN token from APN server
7. APN server returns APN token to iOS application
8. iOS application sends username and corresponding APN token back to Twoefay server
9. Twoefay server adds APN token to corresponding username in database



1. User signs into website using a username and password
2. Website authenticates user via username and password
3. Website requests authentication from Twoefay server using user’s username
4. Twoefay server indexes the database using the username
5. Corresponding token is returned
6. Twoefay server requests push notification from APN server via token
7. APN server pushes notification to users device
8. User interfaces with TouchID and result is sent to Twoefay server
9. Twoefay server responds to webserver with authentication results

The backup authentication flow contains user registration, user login, token generation, device registration, OTP generation, and access approval.



Begin

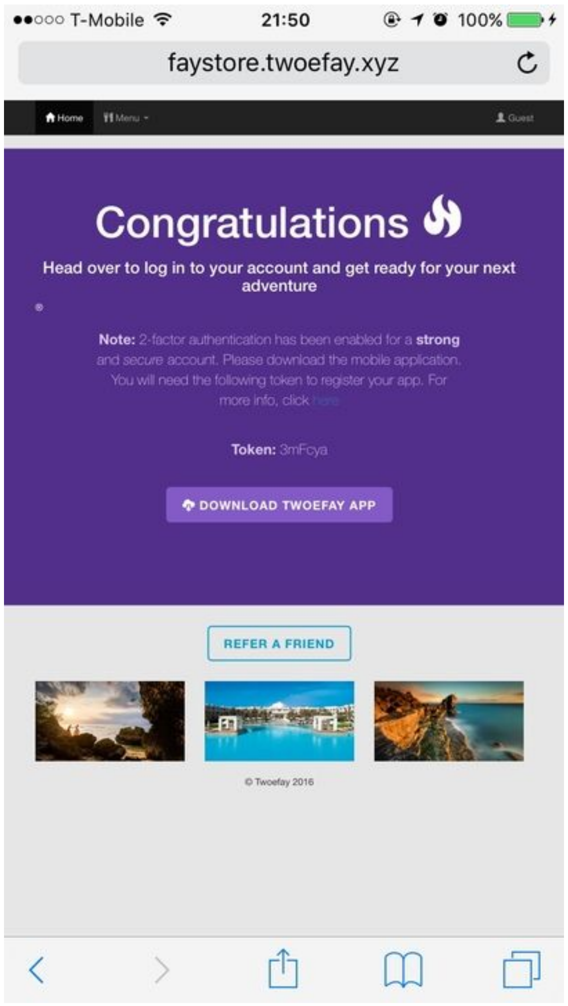
Username test2149

Email wu.mfay@gmail.com

Phone 3104248872

Password •••

*A user registers an account with the Client website.*



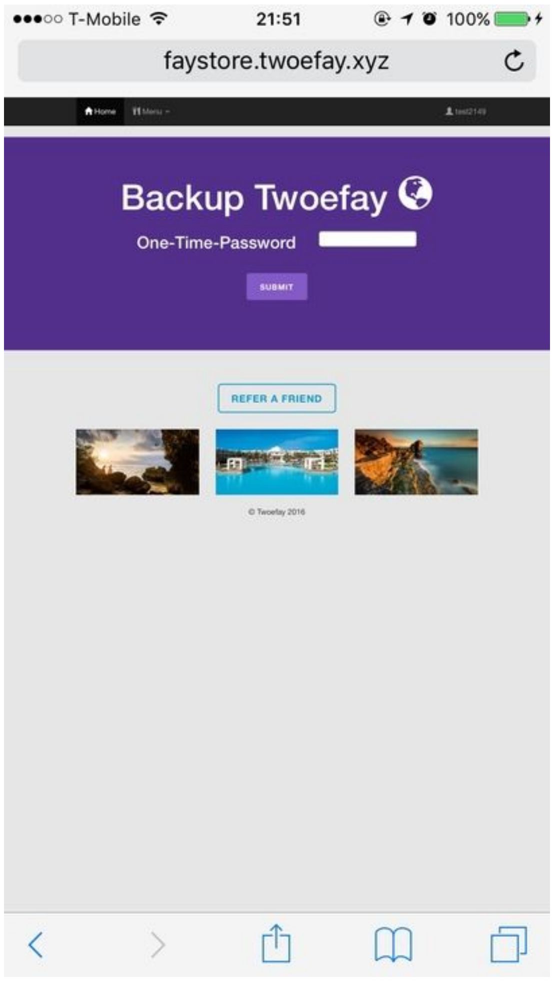
Congratulations

Head over to log in to your account and get ready for your next adventure

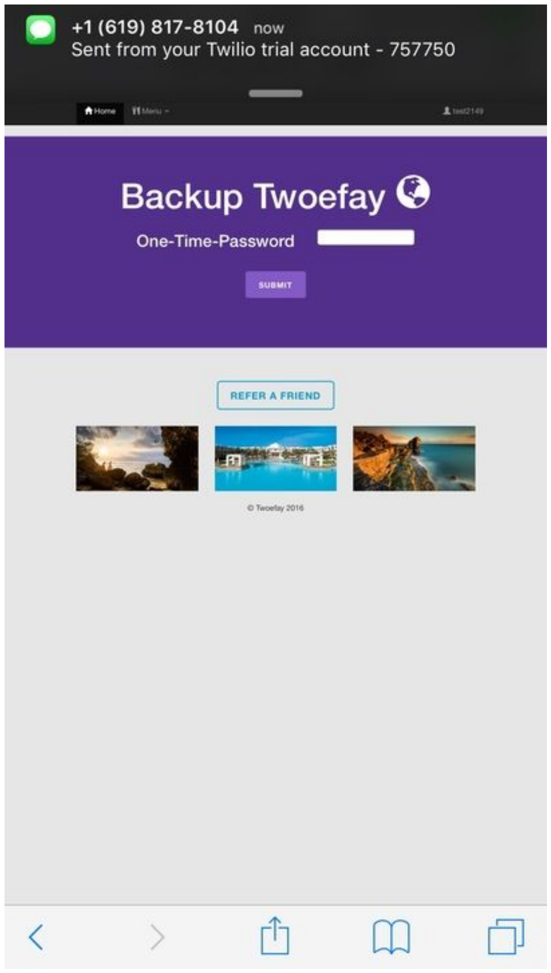
**Note:** 2-factor authentication has been enabled for a **strong** and *secure* account. Please download the mobile application. You will need the following token to register your app. For more info, click here

**Token:** 3mFcya

*The user receives a token which registers their device with the Client through the iOS application.*

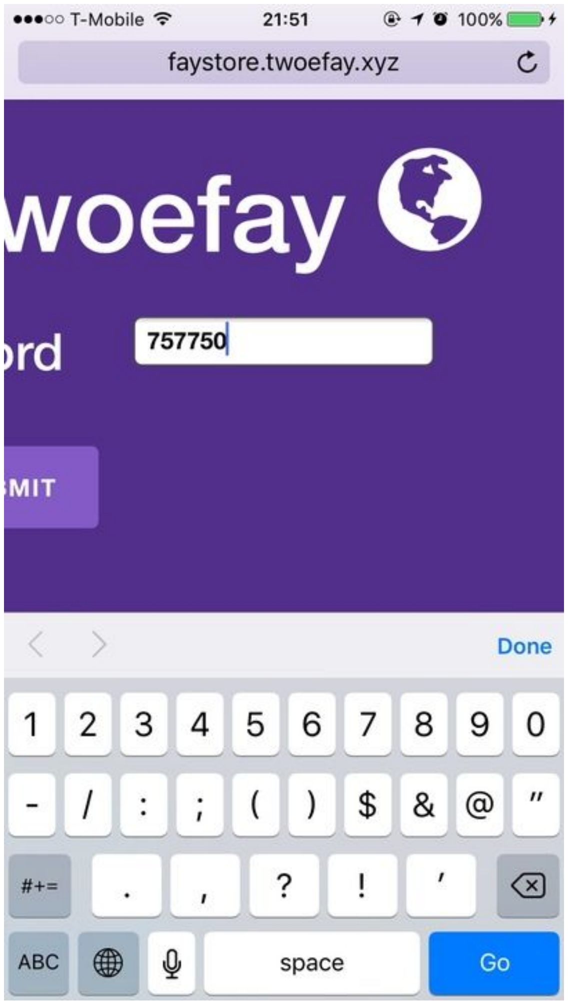


The user waits for an OTP text from Twilio which grants access to the Client website.

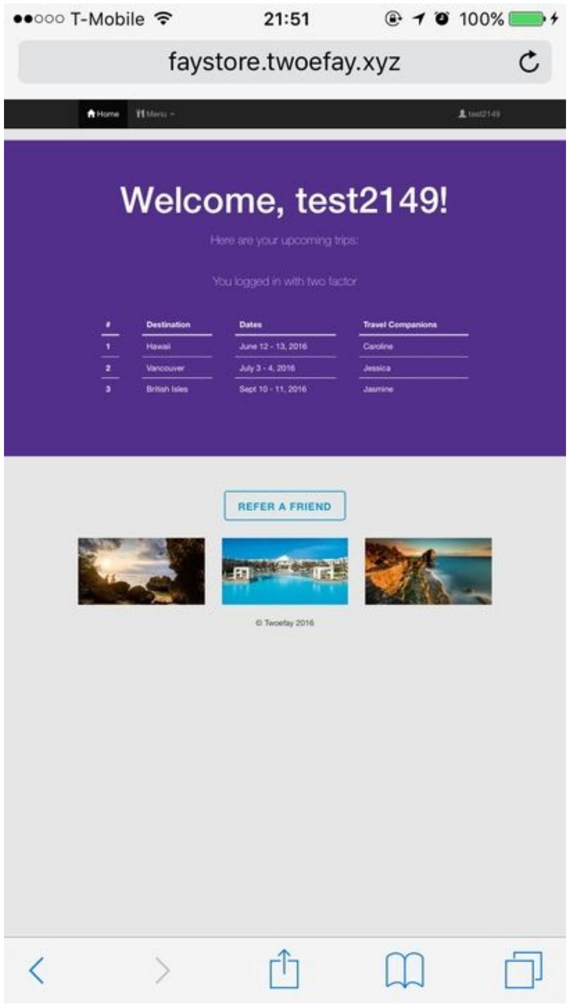


+1 (619) 817-8104 now
Sent from your Twilio trial account - 757750

The user receives the OTP and proceeds to enter it into the text field.



*The user enters the OTP into the text field and proceeds to click Submit.*



Welcome, test2149!

Here are your upcoming trips:

You logged in with two factor

| # | Destination | Dates | Travel Companions |
|---|---|---|---|
| 1 | Hawaii | June 12 - 13, 2016 | Caroline |
| 2 | Vancouver | July 3 - 4, 2016 | Jessica |
| 3 | British Isles | Sept 10 - 11, 2016 | Jasmine |

*The user is granted access into the Client website.*

## Verification/Validation:

demonstrate that each functional requirement or non functional requirement has been met with appropriate screenshots/metrics.

Please See the Security Document at the bottom.

## Challenges:

What were the biggest challenges your group faced? What would you have done differently?

Providing security is not an easy task, if it were it would be easy to circumvent it. Having the additional goals of making it user-friendly and future-proof greatly increase the complexity of design and implementation. We faced numerous challenges in our implementation of the Twoefay system, from implementing our chosen communication protocol, through integration of the 4 components, to SSL certificates.

To future-proof our system, we opted for using HTTP/2 for the Client, Server, and iOS App communication. This was challenging for the following reasons:

**HTTP/2 Issues**

- The HTTP/2 protocol standard is only a year old, making it difficult to obtain information on from a single, trustworthy, source. It is easier to find information on SPDY, an earlier HTTP/2 implementation by Google, than on the most current one. SPDY, however, differs significantly in that it utilizes the simple but less secure gzip for header compression, rather than the newer, faster, and more secure HPACK.

- HTTP/2 is very different from its predecessor, HTTP/1.1. Under HTTP/2, the headers are compressed and data is transmitted in binary, which gives it a significant speed and security advantage but makes debugging it very difficult: unlike HTTP/1.1 you can't simply look at what was transmitted to figure out where the error was, nothing is in clear text. There is no easy way to debug it, you simply have to go back and look at your code, which, as explained in the previous point is difficult because we weren't sure what the correct protocol negotiation and header compression must look like.

- HTTP/2 communication "requires" TLS encryption. While the proposed standard itself does not require it, we found that none of the browsers that have implemented HTTP/2 (which is all major browsers) support unencrypted communication. So while it is not technically required, it was effectively mandatory for us to implement HTTP/2 with TLS encryption. This in itself presented another major hurdle as none of us had any significant prior experience with TLS and at times we were not sure if communication issues were due to faulty HTTP/2 protocol implementation or faulty TLS negotiation.

- After unsuccessfully trying to manually implement HTTP/2 communication for over a week, our team turned to using libraries that already provide support for it, such as Hyper H2 for Python and nghttp2 for C. These libraries allowed us to implement the underlying functionality we needed but not without great effort and numerous issues. Hyper, for example, is still in an early alpha stage of development, as such it has bugs (some not yet discovered) that we've had to work around.

- Much of our initial testing, in particular building up the server and APN communication, was accomplished through the excellent Unix tool cURL. While cURL comes standard on most *nix distributions today, the default version does not support HTTP/2 requests. To address this we had to manually rebuild cURL with the nghttp2 library in our own environments to gain this extra functionality. There are similar tools available, such as httpie and hyper, but we felt they were not as useful for our purposes.

- Differences between Python 2.x and 3.x provided an additional source of frustration. For instance, the SSL libraries in versions older than Python 2.7.9 do not provide support for the NPN extension to SSL/TLS without which HTTP/2 communication is not possible. Having numerous version on our systems caused some issues too, because Ubuntu relies on a specific version of Python and changing the default version broke some basic functionality. To address this issue we opted for working entirely out of Python virtual environments. This ensured we could standardize on specific versions of Python and its respective libraries without damaging our systems.

- Nghttp2 is a low-level HTTP/2 framing library that does not have built in functionality for network IO. This made it difficult to understand and use because there were additionally dependencies on an asynchronous IO library called libevent. The implementation of our apple push notification server abstracted this complexity away by directly invoking the nghttp command line utility included in the Nghttp2 distribution.

**Certificate Issues**

- Generating self-signed certificates is a fairly straight-forward process and while these certificates work fine for testing communication locally, they are not sufficient for enabling communication on a deployed server as they are not recognized by any Certificate Authority.
- When working with desktop operating systems, web browsers, and command line tools, self-signed certificates are not an issue. However, iOS applications are very much a "walled garden." Apple provides a very specific API for iOS apps to access the internet, and the API was giving us issues when attempting to access a website (our server) that had self-signed certificates.
- After trying numerous ways to work around this, including creating our own Certificate Authority, this yielded limited success. Ultimately, to remedy this issue, we turned to the Let's Encrypt Certificate Authority, which allowed us to generate our own officially-recognized certificates for free.

**iOS Push Notification Issues**

- Apple Push Notification cannot be sent from an arbitrary developer to any arbitrary device; Apple maintains strict control over which devices can get Push Notifications from which apps.
- The only member of our development team who has a Developer Account/License with Apple is Chris, however, during development other phones, such as Jon's phone needed to be able to receive Push Notifications in order to test the app.
- Possible Approaches to the issue that we explored:
  - 1. Purchase an organization account through Apple Member Center
    - Costs $99
    - Is fast and would enable everything
  - 2. Share Chris' account on all development machines
    - Option 2 though to be the best because it's the cheapest and fastest.
    - Chris needs to have physical access to development machines to install the necessary credentials.
  - 3. Apple Testflight, their beta testing service
    - This is free, but the service is slow to process new builds and many intermediate steps need to be taken to create a new beta test version (only full versions can be deployed to this service) every time a test is needed on Jon's Phone
- Final Solution
  - Use of iPhone Device UUIDs and Apple Provisioning Profiles generated by Chris online through Apple creates a file that allows Jon's phone to receive Apple Push Notifications

**Client Website Issues**

The challenges with implementing the client website included SSL certificate issues and choosing the right framework for our purposes. Since the Twoefay server communicated over HTTP/2 and TLS encryption, the website needed to send POST requests over HTTP/2 and TLS as well. Although Hyper documentation stated that the protocol would revert to HTTP/1.1 if this is what the communicating party uses, this feature is not working. Thus we chose between bringing all of our communication down to HTTP/1.1, which would defeat our security goals, or figure out a way for the website to send requests over HTTP/2 and TLS to the server. Although not ideal, we chose the latter method and hacked the certificate requirement on the client website end so that it would not require a certificate and instead send over HTTP/2 to the server without issues.

We also faced the challenge of choosing the right framework for our sample website since our team members were largely unfamiliar with web development. After prototyping the website in Scala and Play, pHp, Python and Flask, and Python and Django, we chose Python and the Django web framework for its flexibility and extensive online community support. One of the difficulties encountered no matter what framework was the issue of working with the user authentication libraries which were available for each of the different frameworks. However, these given user authentication libraries did not allow much flexibility with inserting an intermediary step in order to check for Twoefay multi-factor authentication before completing a sign in. We decided to implement the user authentication system without using an existing authentication library and instead chose to do it by using standard Django forms. This can be considered less secure, but served our purposes well as we wanted to demonstrate how easy it would be to sniff user information when it is not encrypted. If we were to create another sample client website given more time, we would further attempt to integrate the Twoefay system with existing user authentication libraries.

## Future work:

what was left out?

Having had only 8 weeks to work on implementing Twoefay, the list of future work is, alas, at least as big as the list of what has already been implemented. The order in which the following are provided is not significant:

- Provide fallback support to the HTTP/2 server. In its current state, the Server will only communicate via HTTP/2, which may not yet be feasible for every website that wishes to use our service, a fallback to HTTP/1.1 would be beneficial.

- Tighten security. In its current state, the Server does not ensure all incoming communication is authorized. While it verifies requests from the Client using its IP, it does not have a similar way to ensure incoming iOS App requests are not sent by a malicious third party. We implemented a way to verify all such limitations on the Server via an AES-encrypted date & time string passed as an additional request parameter, which makes the request time-sensitive and secure. However we did not have enough time to implement the same functionality on the Client and App. There may also be better ways of accomplishing this, such as with mutually-verified certificates, but this will have to be considered in future implementations.

- Improve performance and reliability. While the Server performs as intended, the Twisted framework is not thread-safe and while it does provide ways to implement some threading, due to time constraints we were unable to build a threaded Server. Performance can also be improved by implementing asynchronous communication. The Server code itself can be cleaned up and optimized, as Python was not the native language of the primary developer.

- Add the Twoefay server endpoint for additional backup options such as sending an email with the back-up one time password instead of the text message. The ability to send an email is implemented for the server but we would need to add an endpoint for the client to be able to specify email or text option. Additionally, we are using a trial account for the current text message provider, Twilio, which only allows sending text messages to one verified phone number which restricts the number of Customers we can sign up currently to one. This is clearly not scalable and we will need to either upgrade to a paid account or switch to a different text message provider. Other Customer account management features to be provided in the feature include re-registering an iOS application for a Customer with a different app token, deleting Customers, and disabling Twoefay. Although we highly suggest enabling Twoefay for all Customers, the Client may need or desire the flexibility.

## References:

HTTP/2: Specification: https://tools.ietf.org/html/rfc7540
HPACK Specification: https://tools.ietf.org/html/rfc7541

Known HTTP/2 implementations: https://github.com/http2/http2-spec/wiki/Implementations
Hyper (Python) implementation: http://python-hyper.org/en/latest/
Nghttp2 (C) implementation: https://nghttp2.org/

HTTP/2 vs SPDY: https://http2.github.io/faq/#whats-the-relationship-with-spdy
Let's Encrypt: https://letsencrypt.org/
Flask: http://flask.pocoo.org/
Django: https://www.djangoproject.com/
iOS Security White Paper: https://www.apple.com/business/docs/iOS_Security_Guide.pdf

# CS 130 Twoefay Attacks and Countermeasures

## Access Control and Permissions

### Server

<u>Attack on the MySQL Database</u>

- Possible if
    - Attacker performs an SQL injection to obtain information from, or to corrupt, the database.
- Countermeasure
    - The Client website uses Django's querysets which sanitize user input and prevent SQL attacks.
    - We are doing input sanitation

## Exploits (Buffer Overflows)

### Server, Client, APN Service

One of the benefits of using Python in the Server implementation is that is a type safe language and is thus not vulnerable to buffer overflow attacks.

### App

The iOS app uses the Swift language, which is an improvement in some respects over Objective-C, which itself is somewhat an improvement over C for overflow attacks.[1] We do not use make use of the UnsafePointer construction nor do we try to display strings entered by the user.

[1] https://www.checkmarx.com/2014/08/20/swift-security-issues/

# Man in the Middle Type Attacks

## Server-Client Communication

Attack on /backup, /register, /login

- Possible if
    - Attacker sets up a fake website that looks like Fay's website
    - Attacker spoofs Fay's IP address so 2FA Server accepts the requests as valid
- Countermeasure (not implemented)
    - Have public/private key encryption between the Client and the Server

## Server-App Communication

Direct Attack on /verify, /success, and /failure API

- Possible if
    - An attacker gets the "id_token" for the user they want to attack
- Will Not Help
    - We **cannot** verify the IP address of the user's cell phone as a defense because the IP address will be constantly changing
- Countermeasures (not implemented)
    - We could require that the user use both a one-time-password texted to their phone AS WELL AS the fingerprint, so even if the attacker tries to use

Attacker Overrides dev_token

- Possible if
    - The attacker gets the id_token and configures his iPhone using that id_token
- Countermeasure (not implemented)
    - The dev_token sent by /verify will change very rapidly as both the user's actual phone and the attacker's phone will both be regularly sending a new dev_token to the server
    - Place a check in the database that if the dev_token is changing too rapidly that the account should be flagged

## Server-APN Service Communication

Ideally, the Server and the APN Service should be run on the same device. If they were run on the same device, there would be no room for man in the middle attacks between these two parts of our system. This is currently not the case.

Attacker MITM Server-APN Service connection and Spoofs Information
The payload of the communication from the Server to the APN Service contains 3 pieces of information - what service is being used, what username is being used, and what IP address the request is coming from. If an attacker uses this attack together with knowledge about the user, this can be a fatal attack.

## DDoS Attacks

### App

It does not make sense to DDoS the user's iPhone. If they did, it might block the push notification from going through, then after the 30 second time limit is up, the user is unable to login to their Client website.

### Client

If an attacker DDoS's the Client website, the user cannot even initiate the login process because they will not be able to open the website at all

### Server

If an attacker DDoS's the 2FA Server, then neither the App nor the Client will be able to successfully talk to it.

### APN Service

If an attacker DDoS's the APN Service, then the Server will not be able to provide it information to send push notifications.

## Botnet Detection and Defense

The Server, the Client, and the APN service are on Linux machines that have the potential to be hacked and turned into a botnet devices. As far as we are aware there have not been any known instances of an iPhone participating in a botnet, however researcher have shown that it is possible.[2]

[2] https://www.wired.com/2014/08/yes-hackers-could-build-an-iphone-botnetthanks-to-windows/

# Replay Attacks

## Server-Client Communication

<u>Replay of /backup, /register, /login API</u>

- What would happen?
    - The
- Possible
    - The server does not verify timestamps at all, so replay attacks are possible for any and all of the APIs
- Countermeasure
    - Using a nonce with the public/private key encryption mentioned in previous

## Server-App Communication

## Server-APN Service Communication

<u>Reply of Push Notification</u>

- What would happen?
    - The attacker might be able to spam the user with a lot of repeat notifications for the same login
    - The user might accidentally accept one of these thinking it's legitimate if the attacker does it at the right time
- Possible
    - As mentioned before, Server and APN Service are on different machines, so reply is possible because there is no timestamp
- Countermeasures (not implemented)
    - Best would be to have them both on the same machine
    - Second best would be to have the nonce