# Cryptography White Hat Writeup

Ian Conrad, Franklin Johnson

April 2023

## 1 Introduction

Our entire project implementation was in Python, although some aspects were tested and verified in C++, i.e. prime number generation. Our project is broken up into two scripts, a client and a server. The server is started one machine and then any number of clients can connect to it once is deployed. Each clients session runs on a separate thread, solely dedicated for handling that client. However, the server only allows a single user, to be connected regardless of how many clients attempt to do so with valid user credentials (e.g. John Doe is connected to one machine. John account information cannot then be connected to through any other account). There is a variable start-up delay for the server to connect and generate all session keys. This delay is an expected part of the server implementation. This server implementation also does not have any persistent state information, but rather resetting the server can be done by simply terminating and restarting the server process. That being said, while the server is running, clients have access to their data persisted beyond the an individual client session.

## 2 Usage and Running

We've included our entire project as a zip file. For installing, you need only unzip the file, and install the correct version of Python (see $.python - version$ file) for prerequisites. After collecting those dependencies, the server and client can be set up (on a Linux-based install, mimic as appropriate for Windows) as follows:

1. `mkdir venv`
2. `python3 -m venv venv`
3. `source venv/bin/activate`
4. `pip install -r requirements.txt`
5. `python SERVER_main.py` Leave this running
6. `python CLIENT_main.py` Run this in a new terminal

## 3 Packet Implementation

The packet structure and design is the basis for all the underlying data migration. We describe this here first because we will reference models defined here throughout the rest of our description. To begin with we have two models.

```
ProtectedOperation:
    id: int
    nonce: int
    data: str
    value: Optional[int]

Operation:
    id: int [IdTypes]
    data: str
    mac: Optional[str]
    signature: Optional[tuple]
```

The above show the basic data models for our `Operation` and `ProtectedOperation` classes. We'll begin by discussing `Operation`. This class serves as a super set for all our data transfer. This includes the unencrypted data transfer that occurs during the server-client handshake. The reason for this is so that we can label our data as encrypted or not and also highlights what stage of the handshake process we are at when receiving data from the client.
We reference `IdTypes` in the definition of `Operation`. `IdTypes` is an enumeration of all possible ids that we accept for `Operation`.

```
IdTypes:
    HELLO = 0
    KEYGEN = 1
    ENCRYPTED = 2
```

All of our encrypted data is batched into the same type of `Operation`, that being with `IdTypes=Encrypted`. We intentionally flatten our labelling scheme when compressing from the many types of `ProtectedOperation` to the limited types of `Operation`. This is done so that we cannot derive any information about what type of operation we are seeing just from sniffing the unencrypted `id` label tacked onto the `Operation` representation.

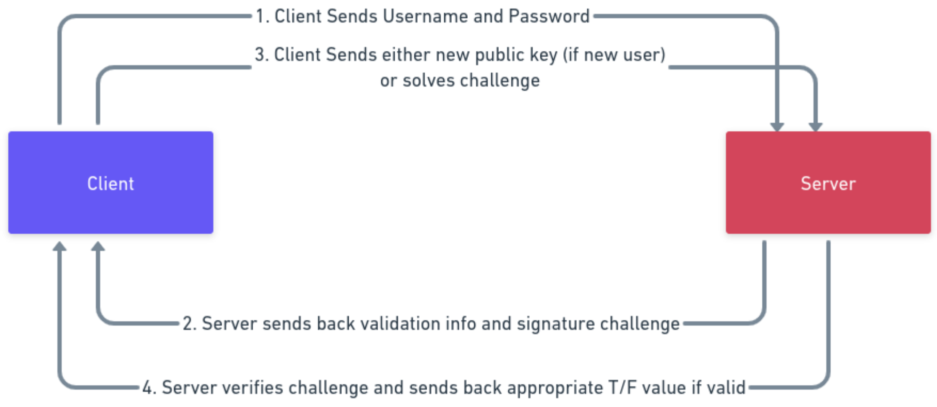
We've talked a lot about our unencrypted `Operation` class without discussing why we even have the `ProtectedOperation` class. We use `ProtectedOperation` as a holder class for all our data, prior to encrypting the package for transfer. Most of the time the data in `ProtectedOperation` is unencrypted prior to being transformed into a `Operation` for network transfer. The exception there is values for incrementing and decrementing the bank values, which remains encrypted in the form of a Pallier cryptocounter. The typical process for transforming a `ProtectedOperation` into a `Operation` is as follows:

1. Format raw data into `ProtectedOperation` with data being a string representation of a payload
2. Generate a Nonce to attach and flag the `ProtectedOperation` with the appropriate operation being one of the allowed ids (shown below)
3. Take the JSON string representation of the `ProtectedOperation`
4. Encrypt this string with our session key
5. This encrypted string becomes the data field of a standard `Operation` class
6. Generate and attach a MAC and signature for the data before sending to the server for the `Operation` class
7. Label the `Operation` as having `IdTypes=Encrypted` as it contains encrypted data and show be processed as such

We mention labelling the `ProtectedOperation` with the appropriate values. These are all the allowed values, which indicate what exactly the client or server is requesting from each other.

```
ProtectedOperationIds:
    DEPOSIT = 0
    WITHDRAW = 1
    CHECK = 2
    VERIFY = 3
    FREEZE = 4
    CHALLENGE = 5
```

## 4 Life Cycle and Server Operation


1. Client Sends Username and Password
3. Client Sends either new public key (if new user) or solves challenge
Client
Server
2. Server sends back validation info and signature challenge
4. Server verifies challenge and sends back appropriate T/F value if valid

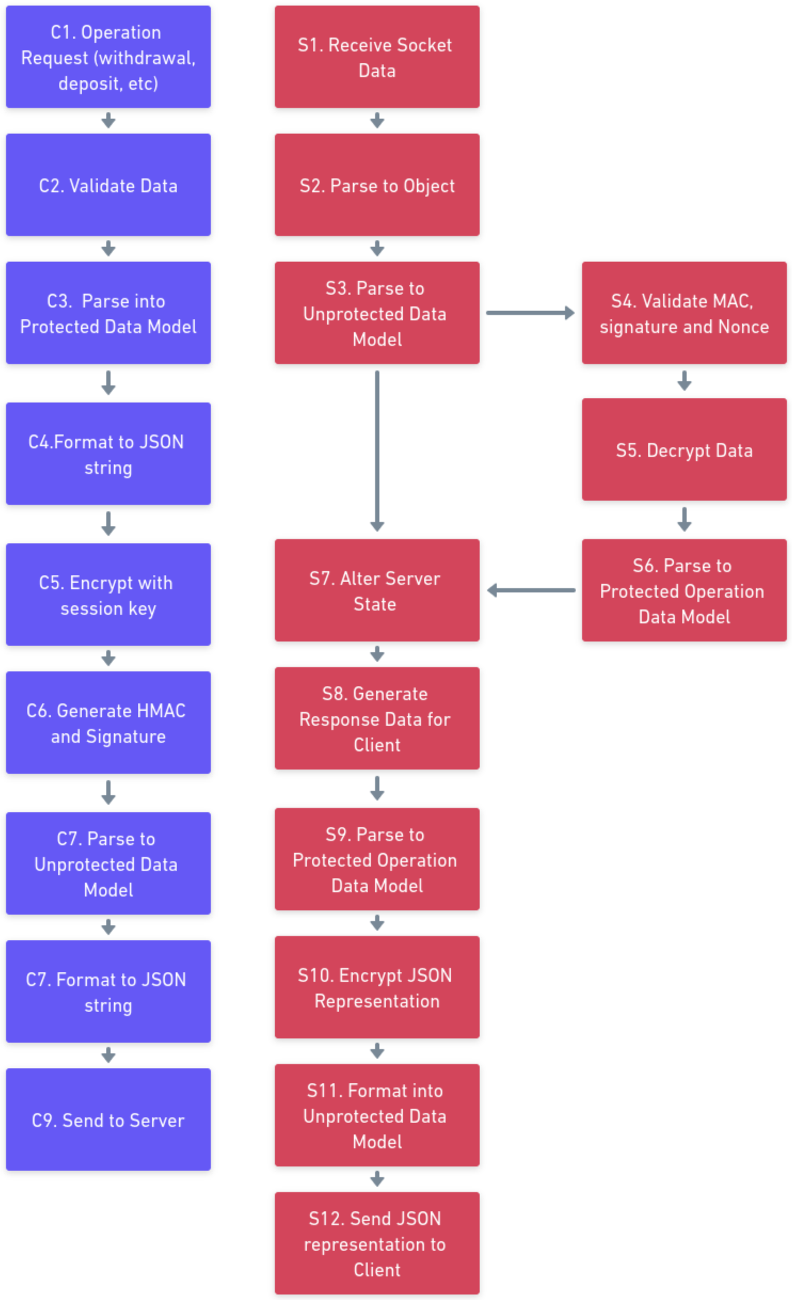
The above image shows a brief overview of how the client authenticates itself with the server. This occurs after the server establishes a session key via TLS. We include a more elaborate description corresponding to each of the above labels:

1. The Client prompts the user for a username and password. This is naively sanitized and then sent to the server as an encrypted operation.

2. If the username and password are as expected or if this is a user that is new to the server, the server sends back a challenge verification. However, if the username and password do not match the expected username and password stored on the server side or if we already have a client instance logged in with the same credentials, the client receives a negative response.

3. Depending on the challenge received, the client either sends a new public key used for signature verification or signs the challenge and sends it back to the server

4. The server takes the challenge value or the new public key and compares it to the expected value if the user already had a public key on file. If the value is as expected or the server ingests the new public key, the server sends back a valid response, otherwise the server sends back a negative response.



C1. Operation Request (withdrawal, deposit, etc)
C2. Validate Data
C3. Parse into Protected Data Model
C4.Format to JSON string
C5. Encrypt with session key
C6. Generate HMAC and Signature
C7. Parse to Unprotected Data Model
C7. Format to JSON string
C9. Send to Server
S1. Receive Socket Data
S2. Parse to Object
S3. Parse to Unprotected Data Model
S4. Validate MAC, signature and Nonce
S5. Decrypt Data
S6. Parse to Protected Operation Data Model
S7. Alter Server State
S8. Generate Response Data for Client
S9. Parse to Protected Operation Data Model
S10. Encrypt JSON Representation
S11. Format into Unprotected Data Model
S12. Send JSON representation to Client

In the above, we document the basic life cycles for when a client sends out a packet and when a server receives a packet. Client icons are represented by purple/blue, and the Server are represented by red. Below is brief description of both the client and server operations. A fair bit of this is repeated information from our Packet marshalling section, but in the context of our server operations for clarity.

## Client

The numbers in the list below correspond to the **Cx.** labels in the diagram

1. The user selects an operation and the client prompts them for information if appropriate. For standard operations, this only happens for withdrawing or depositing money, where the client needs the user to provide information related to how much money should be transferred.

2. We do some preliminary data validation in order to make sure we aren't submitting bad or malformed data to the server.

3. We take our data and format it into one of our protected operations. We have 6 protected operation types (Deposit, Withdraw, Check, Verify, Freeze). The operation type selected entirely depends on the operation selected by the user. We also attach a Nonce in this stage.

4. After generating our protected model, we format this model into a JSON string.

5. This JSON string is then encrypted with our session key using RC5.

6. For this encrypted data, we generate a signature and mac using RSA and HMAC respectively.

7. We take the encrypted data, MAC, and Signature and format them into our unprotected data model, flagging that this is encrypted data.

8. We then format this unprotected data model into a JSON string in preparation for sending to the server via sockets.

9. Finally, we send this string to the server.

## Server

The numbers in the list below correspond to the **Sx.** labels in the diagram

1. All of our server responses are event driven since the server is always listening for new connections. The initial step is simply receiving an string of data

2. Data is transferred in the form of a JSON string, so the first thing we need to do is parse our JSON to a Python dictionary.

3. All of our data from the client to server is wrapped in an unprotected data model, which allows us to identify if the packet is encrypted or if the packet is one of our initial handshake protocols, which should be done in public view.

4. We then validate the MAC, signature and nonce to ensure that the data hasn't been tampered with, is from the intended client and is a unique message.

5. If all these conditions are true, we decrypt the data using the session key established with the expected client

6. After decrypting this string, we have a Protected Operation Model format object and we parse it into our model

7. This cell is an umbrella summary for all the operations that we do dependent on the operation we receive. For instance if we see a withdraw operation, we modify our Pallier encryption representing the balance of the client accordingly. This encapsulates almost all the operational logic of our server. This handles all the enumerated operations for Protected Operations we list previously, any non-standard Protected Operation type is ignored (although it is actually impossible to get through validation without a proper id type)

8. After parsing and adjusting the server state, the server might respond with a packet for the client. This comes in the form of a data string. For instance, for checking balance, the server responds with the balance for the client. The checking balance option is the only operation that ever decrypts the Pallier ciphertext representing the current value of the client's balance

9. The data is then attached to a Protected Operation Data model, with appropriate identifying id. This id matter strictly less to the client than the server since the client blocks while waiting for the server.

10. We then take this whole data model and turn it into a JSON representation before encrypting it using our session key.

11. We then wrap the encrypted data in a Unprotected Operation model for sending to the client

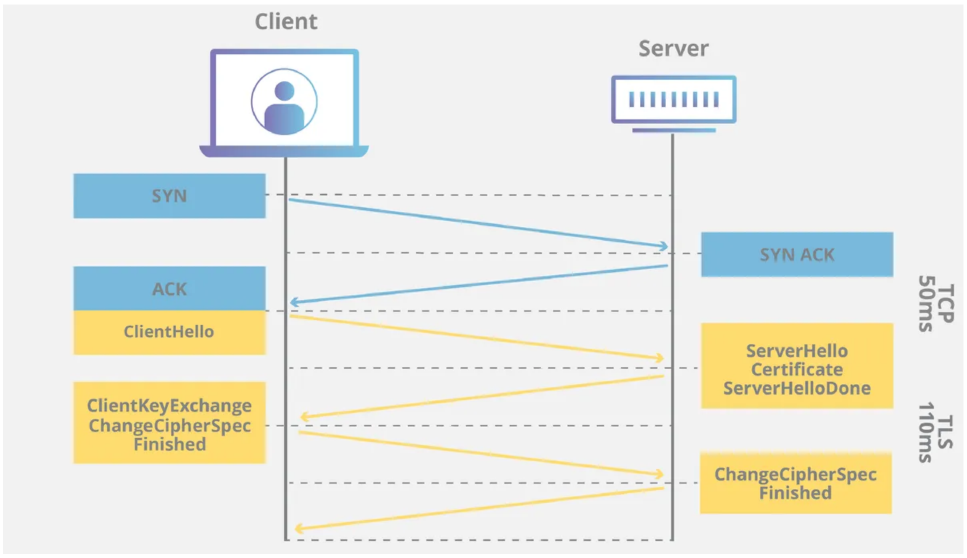
12. Finally, we convert that model to a raw JSON string and send that back to client

## 5 Design Principles

In order to enforce and reduce how malleable our data is, we enforce strong data validation for our packets. This form of schema heavily reduces how mutable

our socket handler methods are. We also had some standard practices when designing this product, such as working on each segment of code in isolated branches testing as we went. When it came time to put it all together, between the server and client sockets, we did this in one fell-swoop from a single source of truth branch.
One of the most costly principles we enforce is separation of clients. No two clients share any values in any step of our encryption processes aside from the initial handshake where each client encrypts the premaster secret with the server's public key. This causes onboarding of a new connection to be expensive in terms of timing, but also ensures that if a single customer is compromised, it will take equal effort to break any other customer and we do not have larger area of effect.

## 6 TLS Implementation



Our server and clients communication is initiated by a handshake resembling TLS. In TLS, the client initialize the handshake by sending a request to the server, with its available cipher and protocol. The server then responds with the protocol, it request the users uses, and the Client complies. Yet in our implementation since we've only created one encryption model, the selection process is include to demonstrate how the values would be selected int the real world. Another slight variation in our model was the process of sending and receiving certificates from a certificate authority. Again, in a normal TLS handshake, the server would authenticate itself to the client using a certificate that the client would verify with a certificate authority. Given the scope of the project, the certificate authority was assumed to just be inherent knowledge in every client, and instead we chose send back just the public key data in an un-encrypted form.

## 7 Symmetric Key Cipher: RC5

RC5 was a natural choice for our symmetric key cipher implementation. It has a simple implementation, strong encryption even with shorter keys, and is highly modular. The only difficulty in implementing RC5 comes from Python integral capacities and wrapping. In C and C++, a datatype of a given length of bits (character, unsigned int, etc) has natural wrapping when exceeding possible values. Python does not implement this same capacity for numbers and encodes negatives in an entirely different format than C. Since Python does not enforce length constraints on its integers, we added packing to C types to handle negatives and integral truncation. After implementing this, the rest of the implementation was as simple as chunking and parsing our keys and texts as appropriate.

For our RC5 parameters, we use $r = 20, w = 16$ so 20 rounds of shuffling of our keys and words of 16 bits. This is fairly standard, although bumping from $r = 12$ to $r = 20$ was purely done as a recommendation for compensating for a smaller key, given Python is prohibitively slow for generating primes anywhere near 1000 bits.

## 8 Homomorphic Cipher: Pallier

We implemented a Pallier cryptosystem as our homomorphic cipher. Pallier made a lot of sense to implement as it is comparably easy to generate new values for given a decent prime generation algorithm. We could have also implemented RSA or ElGamal in homomorphic setups, but those are homomorphic under multiplication, so not directly practical for our purposes. The Benaloh cryptosystem was also a strong candidate here, but we went with Pallier as its modular operations were already supported by our utilities when implementing it. We used a additive homomorphic encryption so that we could maintain encrypted values on the server side and never need to worry about exposing the values unless explicitly decrypted by the server at the request of an authenticated client. We generate a new set of Pallier values for each client such that each client has its own prime values and generators. This is slightly costly, but helps maintain separation of client interests.