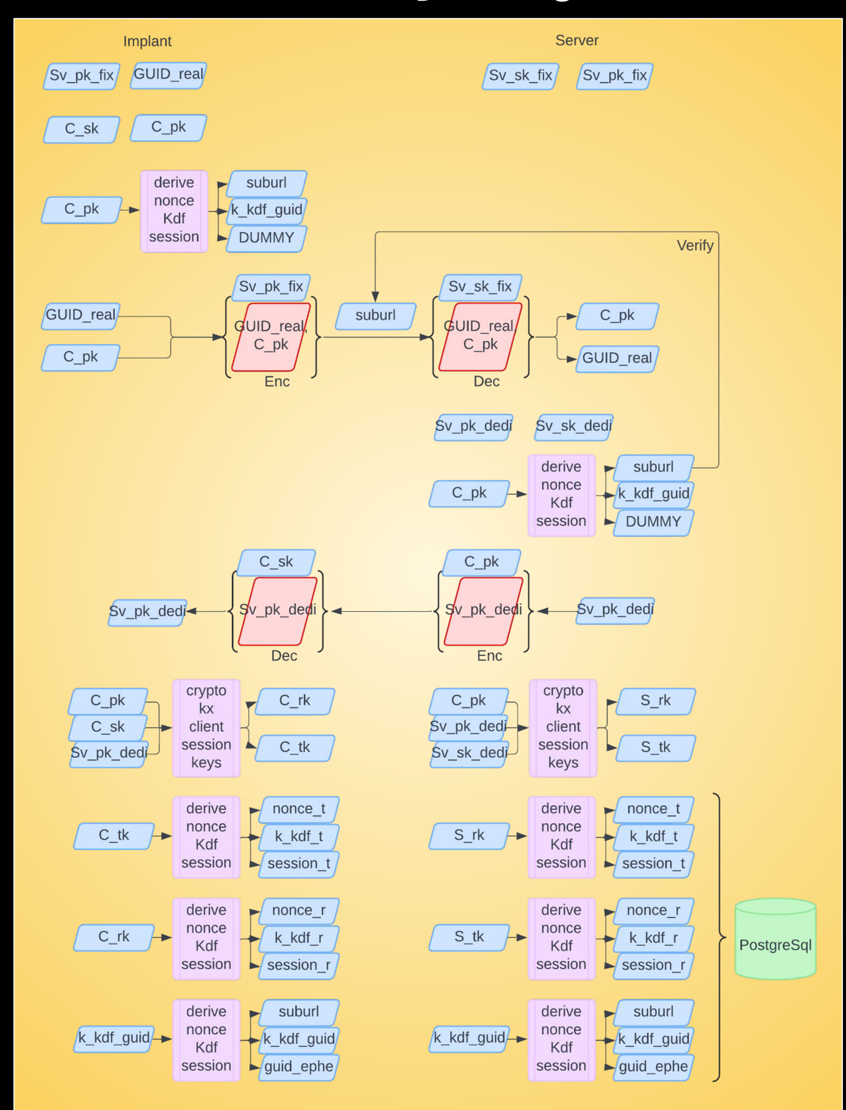
CS6983 Malware Development Capstone Project
Internal Network Penetration C2 Framework

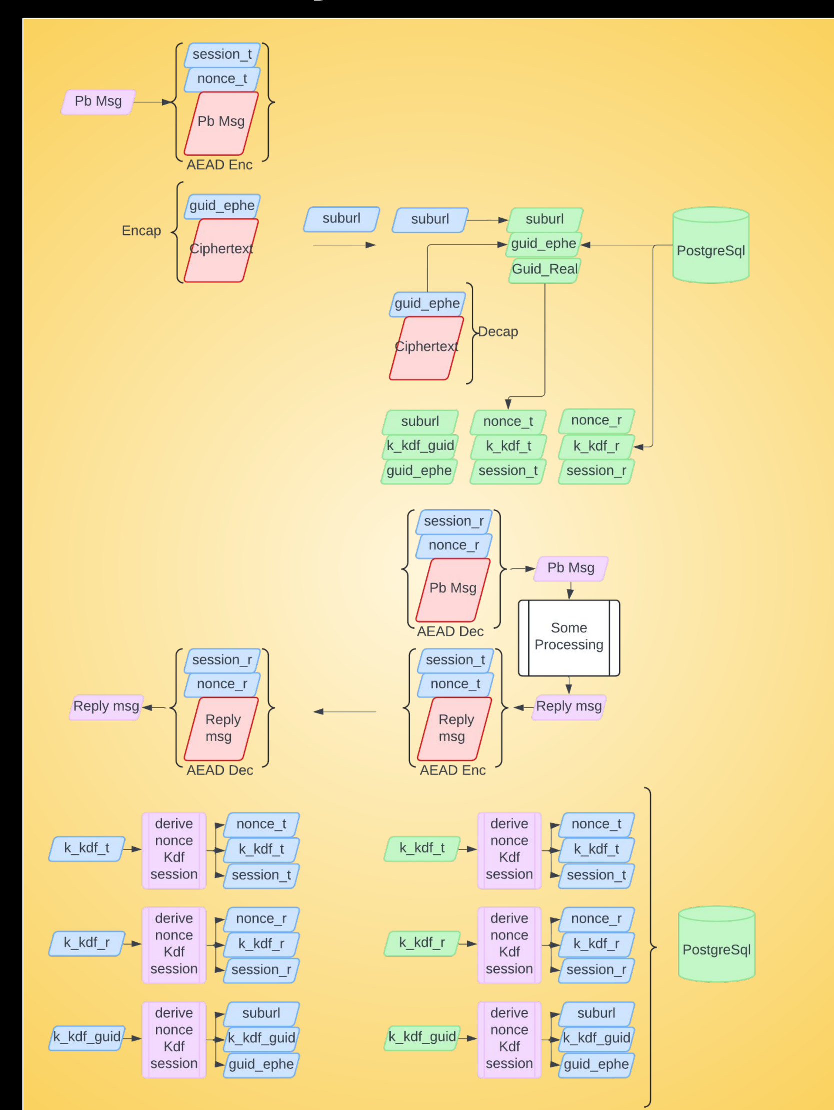
Double Ratchet Encryption

Any traffic between the Server and Implants is encrypted using Libsodium **AEAD**, ensuring **confidentiality, authenticity, integrity** and **forward secrecy**. The **symmetric keys** for encryption will be dynamically generated from 3 root keys each session, and a new **Root** symmetric key will be negotiated every 10 sessions.During C2 communication, there will be unencrypted **metadata** and **URLs** for C2 to decide which key to decrypt the traffic. The **metadata** and **URLs** are randomized each time the traffic is sent, making sure there is no informative data to identify the malicious content from other network traffic for IDS or manual inspection.

Initial Key Exchange



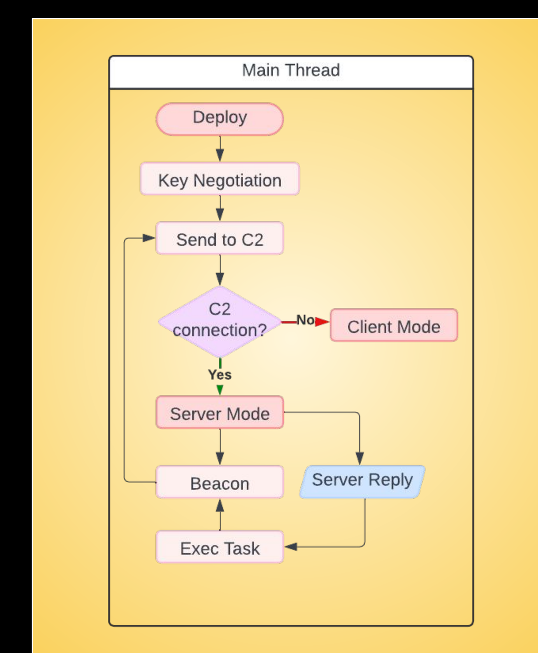
For every connection session



Communication Proxying

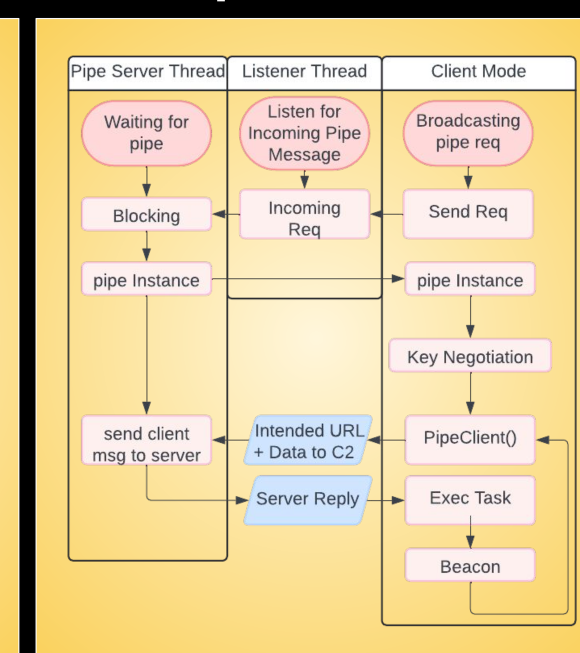
Most of the time, the **network segmentation** and **firewalls** in an organization prevent unauthorized traffic flowing out of the network, making the implant connecting to C2 impossible. To ensure our implant is still functional under network segmentation, our implant is **capable of proxying traffic from other implants** that can't connect to server with **Named Pipe**. Our server and backend are built with this in mind too, it **supports** the control of **multiple implants** connections simultaneously.

Main Thread



After deployment, the implant will start key negotiation and trying to connect with the C2 Server. The connection status will determine whether to enter **Pipe Server** or **Pipe Client mode**. If the connection is successful, the implant will keep going with the server mode and starts to sending the beacon() message until it meets a condition where the server connection is failed.

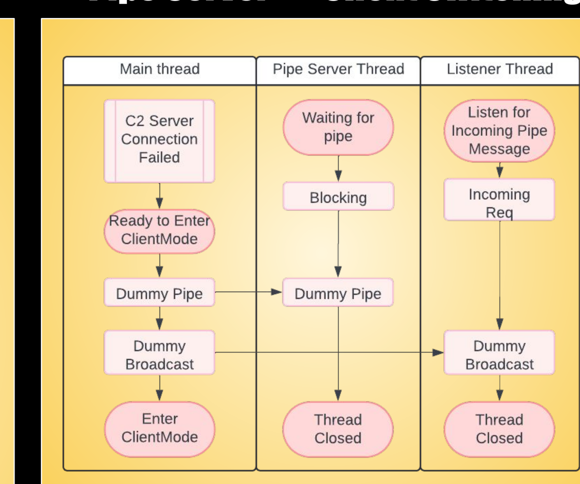
Pipe Server/Client



In **pipe server mode**, the implant consists of 3 threads. The main thread, listener thread and the pipe thread. The main thread executes the main functions, and the listener thread will keep listening for incoming pipe request. The pipe thread will remain stuck until there is a valid pipe broadcast message.

In pipe client mode, the implant will send out a broadcast message to port 6983 of all interfaces. Once the server picks it up, it will keeps the normal operation as if it has network connection.

Pipe Server → Client Switching



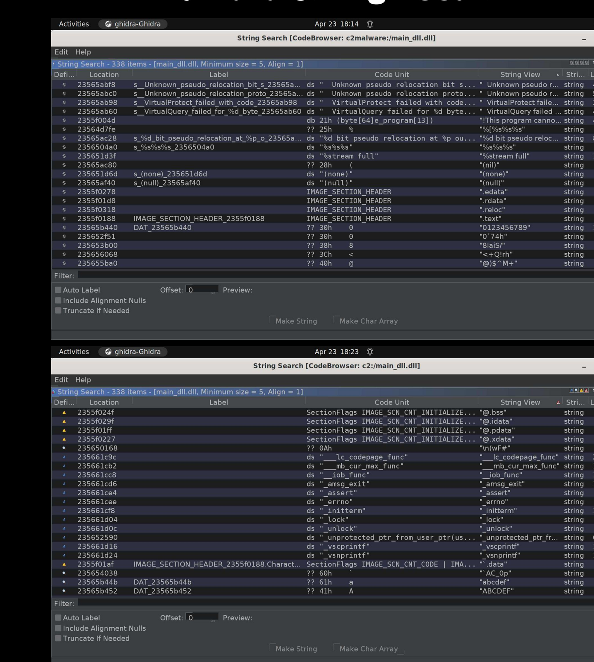
For the implant to gracefully switch between **pipe server / client mode**, the main thread will send out a **dummy pipe** and **dummy broadcast** to its own **Pipe Server** and **Listener thread** to terminate them.

However, there is one problem with the **pipe client mode**. Since the **endpoint URL** is randomized each time for each implant, the **pipe server** does not know which URL it needs to send for the pipe client. Therefore, the implant has to append the intended URL before the encrypted message for the pipe server. If the adversary is able to intercept the message between the pipes and stop it from reaching out to server, record the first 256 bytes and start a DOS attack, they will have a higher chance of success since now our server is expecting that URL.

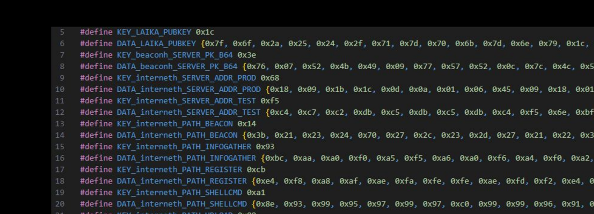
String Obfuscation

To **prevent reverse engineering** to our code, we added **string / code obfuscation.** Each string is encrypted with a **different encryption key**, and will be decrypted on spot by an inline function when being used. Although all data (string and key) will be stored in global data memory, there is no way to distinguish between the strings data and the key just by looking at the global data section. The keys are randomly generated with every build.

Ghidra String Result

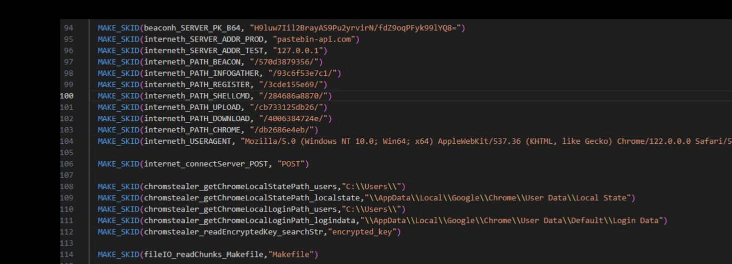


From the string result of decompiler Ghidra, we can see there is no useful string presented in the window.

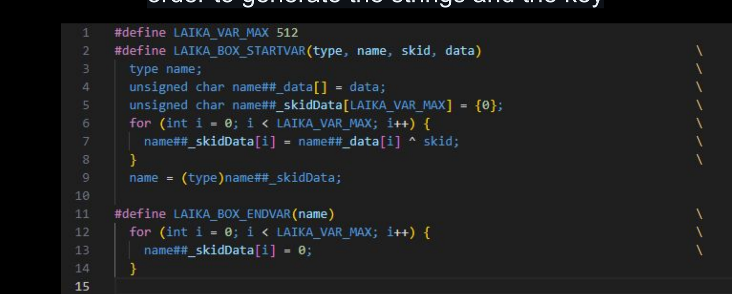


Here is how we include the necessary strings we need in the code. We include them in a header file, as encrypted Binaries.

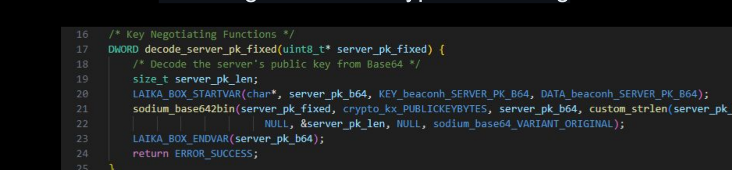
Example Use Case



To start obfuscation, we first need to include the strings we need to a generator file, in order to generate the strings and the key



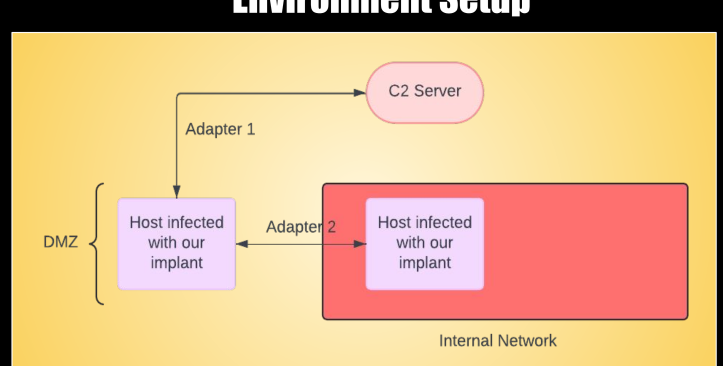
Inline functions being used in the code to decrypt and torn strings in the data section. It's using XOR to decrypt each string



An example of decrypting server's fixed public key

C2 Framework Basics

Environment Setup



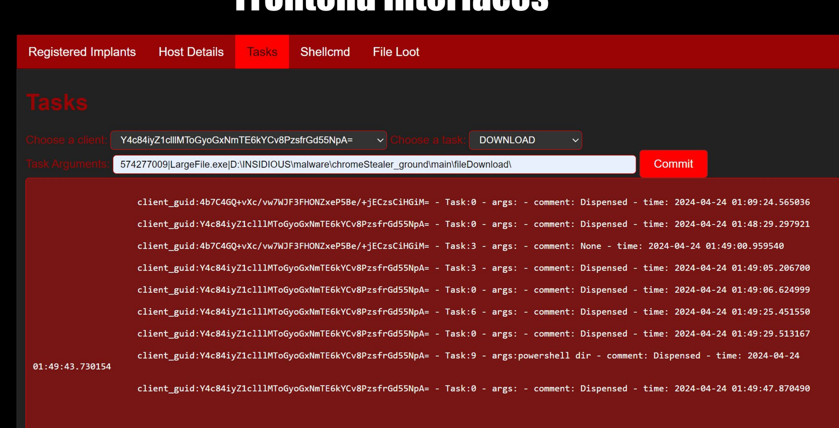
Key Concepts

1. **Client_guid**: This is the unique identifier for each implant. At deployment, the implant will generate a unique ID to let server identify itself. It remains **unchanged** during the implants' lifetime
2. **Beacon message:** It's the message that will be regularly sent to the server in a fixed amount of time interval. It has 2 purposes: tell the server that the implant is still alive and retrieve the task it needs to perform from the server.
3. **Encapsulation**: To prevent information disclosure, the implant will encapsulate any meaningful data inside another message. The final message exposed on the internet will have these two fields: 1. client_guid_ephemeral, 2. ciphertext. Note client_guid_ephemeral is different from the original real guid and will be changed each time the communication happens.
4. **Ratcheting:** The server and the implant will use the same algorithm to generate symmetric keys. Therefore, as long as the root key is the same, any keys generated afterwards will be the same. Each time the implant connects the server, a new ratchet will be performed on both server and the implant.
5. **session_key_x:** Each implant will have 3 symmetric keys. session_key_send, session_key_rcv and client_guid_ephemeral. session_key_send : encrypt any information goes out from the implant session_key_rcv: decrypt any information goes in to the implant (i.e. from server)
   client_guid_ephemeral: Used to authenticate the encapsulated message, changing every time to avoid detection
6. **nonce_x:** Apart from session keys mentioned above, **AEAD** needs an additional data to authenticate and ensure the integrity of the data, a.k.a the nonce. These will be generated dynamically along with the session keys as well. (nonce_send, nonce_rcv) Besides the sending and receiving session keys, a nonce-like data called "suburl_bin" will also be generated along with the **client_guid_ephemeral**. The suburl_bin will then generate a random URL string, then send it to the server. The server will only accept the URL associated with the client_guid_ephemeral .
7. **key_kdf_x:** This is the key used to generate symmetric key and client_guid_ephmeral for each communication. Each implant will have 3 different kdf keys to generate 3 different symmetric keys.

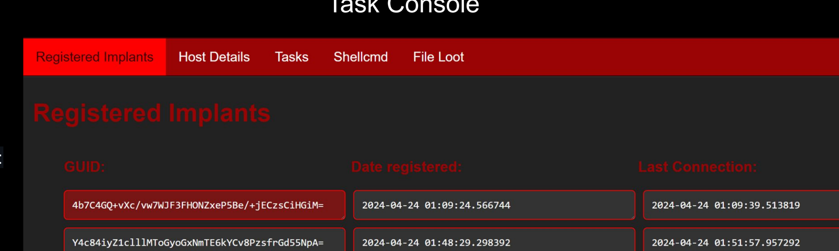
Potential Improvement

- The frontend can be more powerful than the current, for example adding more realtime update.
- The Download file task arguments is too long, we can add a feature to make it simple
- The Shell command execution will wait indefinitely if executing a command that never returns, causing the implant waiting for the thread to complete indefinitely. Adding a killswitch for is helpful
- Our dropper currently only loads the main function into the current process, to add make our implant more stealth, we should consider adding it to different processes
- A different C2 channel should be considered instead of communicating with the C2 directly

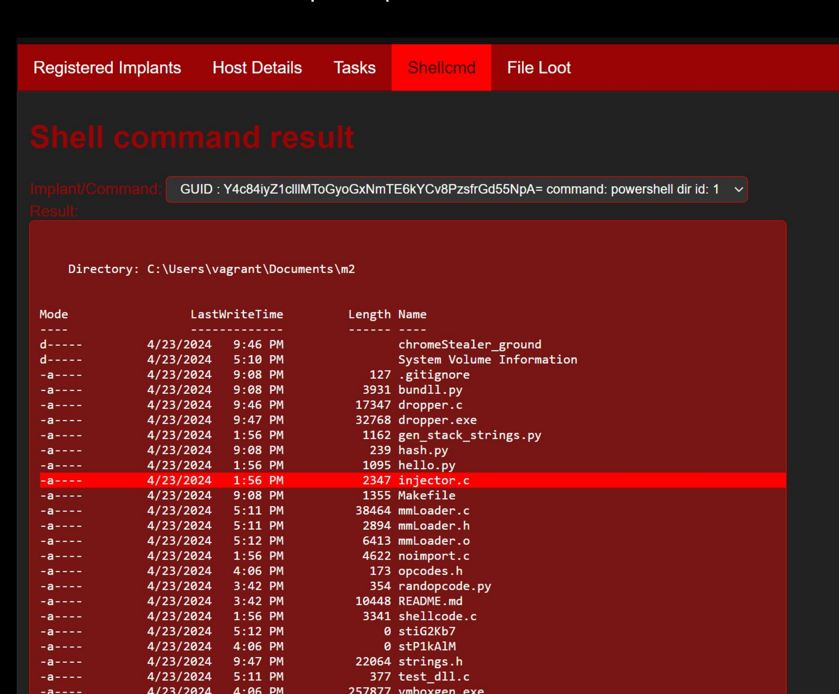
Frontend Interfaces



Task Console



Implants pulse Monitor

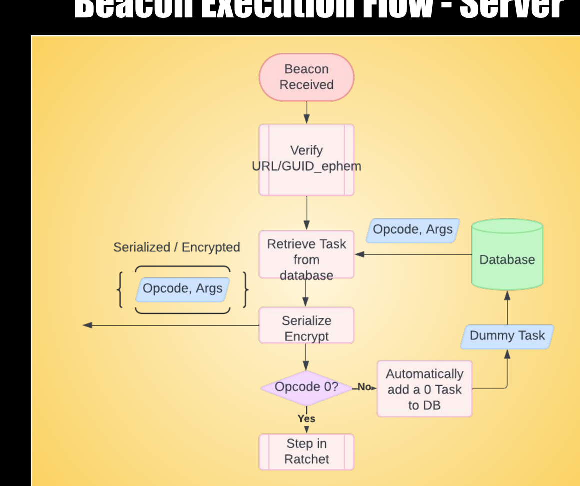


Shell command display

Implant ↔ C2 Interaction

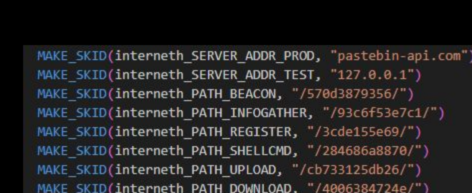
In order to gain interactive access, we have built a backend server using **Flask**, along with a database with **PostgreSQL**. To satisfy the extensive utilities of our malware, we need a Remote Procedure Call protocol. Since most people are using JSON, we decided to use **Protobuf** as our solution. **Nanopb** has a great lightweight C implementation for this.

Beacon Execution Flow - Server



/beacon

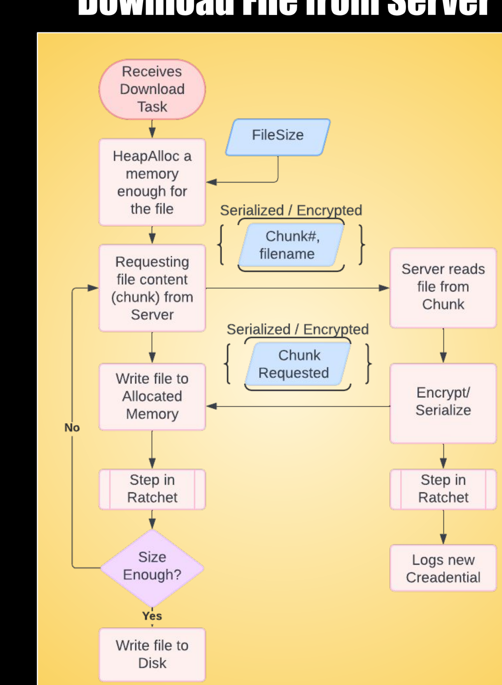
The server logs the beacon message from the implant, submitted via the admin frontend, send the Opcode and arguments to the implant, and prevents the implant to execute the same task



Server Endpoints for implants: These APIs are all randomized to prevent Directory Brute forcing!

- **/register**
  Logs the implant, send PK to implant
- **/beacon**
  Logs the beacon, send latest task
- **/infogather**
  Records the host information from implant
- **/shellcmd**
  Receives the shell command result
- **/upload**
  Receives the chunks of file uploaded by the implant
- **/download**
  Send the chunk of the file requested by the implant
- **/chrome**
  Receives the key and chrome db file and decrypt it.

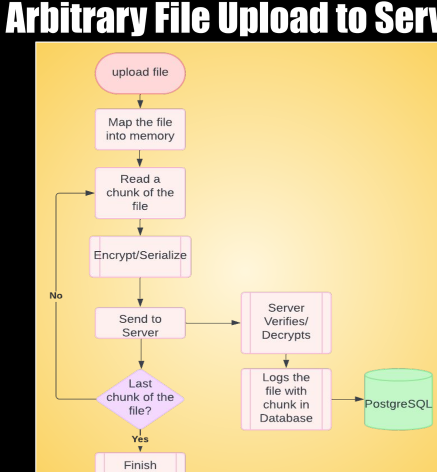
Download File from Server



/download

As you have/will see in the demo, we need to include the file size, filename, and the targeted directory as arguments for the implant

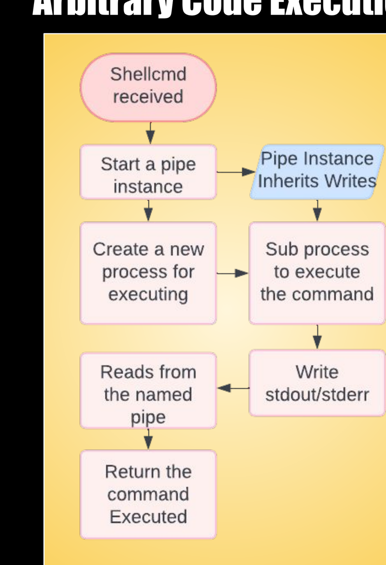
Arbitrary File Upload to Server



/upload

The upload function is almost the reverse of the download function

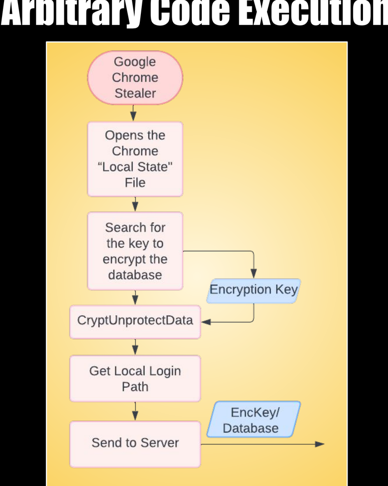
Arbitrary Code Execution



/shellcmd

To execute powershell command, the implant will start a pipe instance and give the write end of the pipe to a new sub process. After the command is executed, the sub process is terminated and the results will be sent back to the server

Arbitrary Code Execution



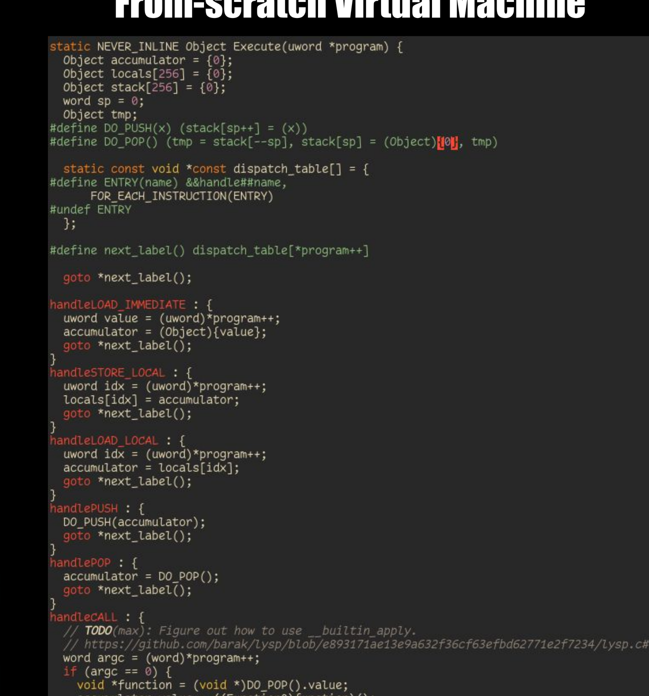
/chrome

The chrome stealer feature is probably the most straightforward. We just retrieve the encrypted encryption key and decrypt using Windows API, then send it along with the database to the server to decrypt

Code Obfuscation

In addition to the string obfuscation, we also implement a **virtual machine** to achieve code obfuscation. Our VM has 10 opcodes and contains a stack, local variables, control flow, and can call arbitrary C functions. To avoid fingerprinting, we generate a randomized opcode table (and therefore randomized interpreter) with each build.

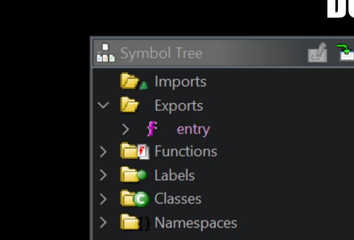
From-scratch Virtual Machine



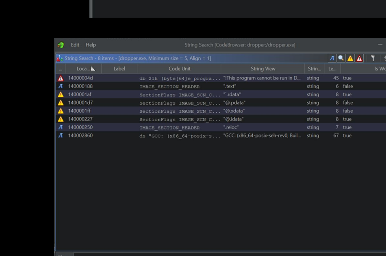
Nothing on disk

Instead of using **URLDownloadToFile**, we download the file to an in-memory buffer and thereby avoid the implant **DLL** ever touching disk. The dropper uses **mmLoader**, an in-memory PE loader.

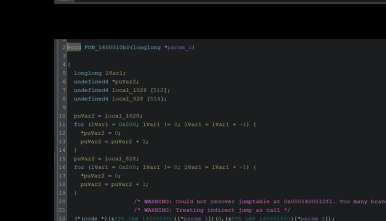
Decompiler Result



No imports listed in the imports table; we manually look through PE export tables for the functions we need.

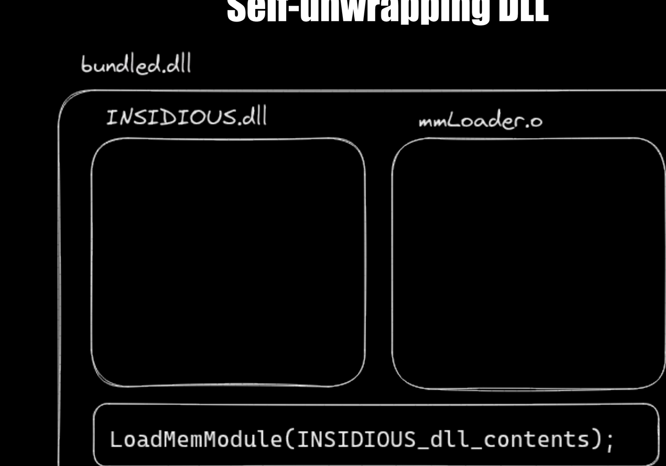


No strings listed in the string window; they are all obfuscated or elided completely in favor of hashes.



The decompiler just gives up. It's all unpredictable indirect tailcalls all the way down.

Self-unwrapping DLL



We wrote **bundl.py**, a program to take an existing DLL (say, our malware) and an in-memory PE loader (mmLoader), and put them together in a new DLL. This new DLL has no imports and doesn't appear to do anything but exposes an entrypoint that unpacks the inner DLL when called.