FP – Design Document

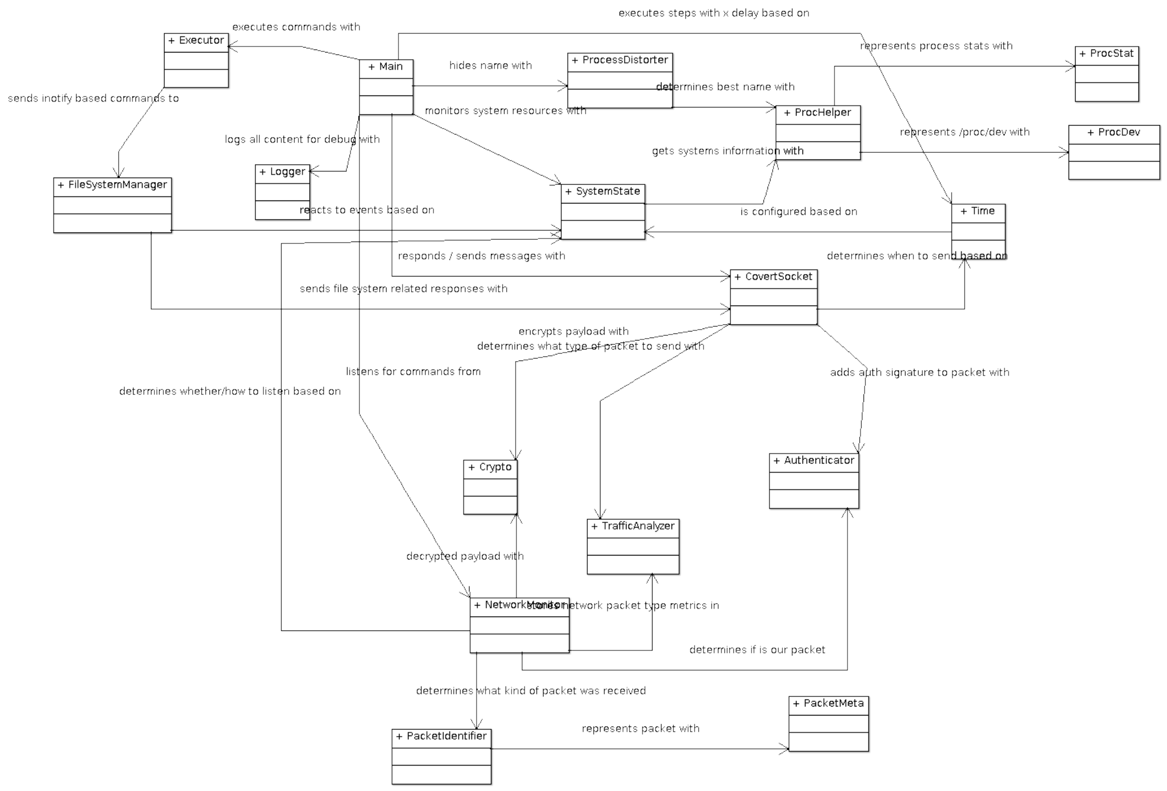
Ben Soer

A00843110

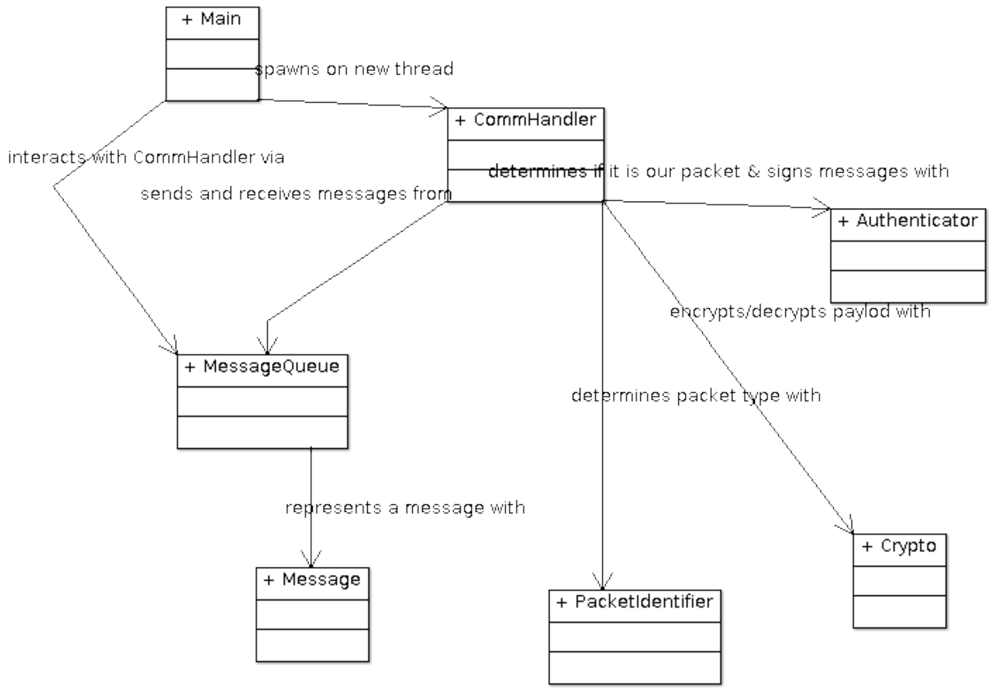
Table of Contents

Class Diagrams

Haanrad (ExFiltration System) Class Diagram

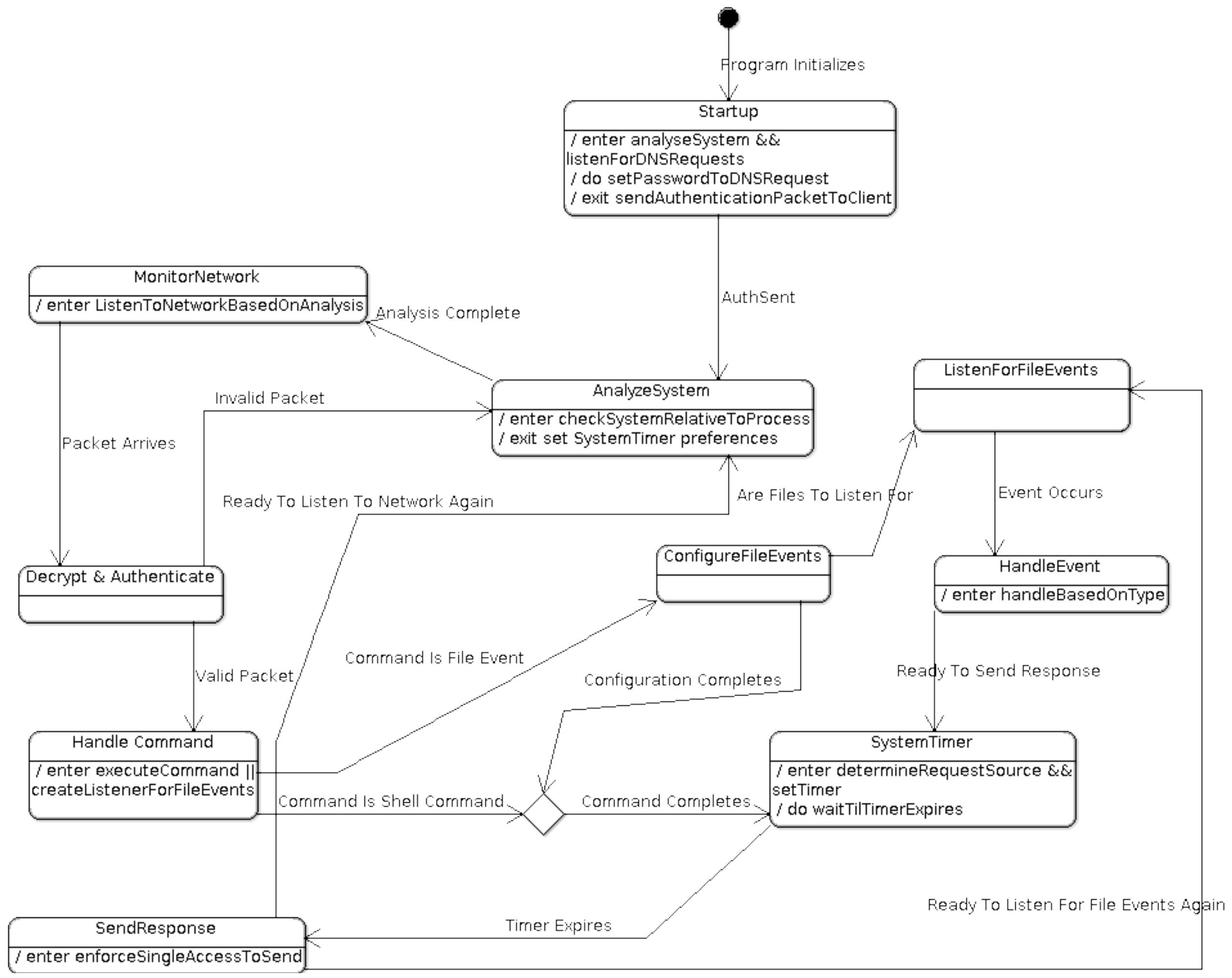

+ Executor
+ Main
+ ProcessDistorter
+ ProcStat
+ ProcHelper
+ ProcDev
+ FileSystemManager
+ Logger
+ SystemState
+ Time
+ CovertSocket
+ Crypto
+ Authenticator
+ TrafficAnalyzer
+ NetworkMonitor
+ PacketMeta
+ PacketIdentifier
executes steps with x delay based on
executes commands with
represents process stats with
hides name with
determines best name with
sends inotify based commands to
monitors system resources with
represents /proc/dev with
logs all content for debug with
gets systems information with
reacts to events based on
is configured based on
responds / sends messages with
determines when to send based on
sends file system related responses with
encrypts payload with
determines what type of packet to send with
listens for commands from
adds auth signature to packet with
determines whether/how to listen based on
decrypted payload with
stores network packet type metrics in
determines if is our packet
determines what kind of packet was received
represents packet with

Client Class Diagram


+ Main
spawns on new thread
+ CommHandler
interacts with CommHandler via
determines if it is our packet & signs messages with
sends and receives messages from
+ Authenticator
encrypts/decrypts paylod with
+ MessageQueue
determines packet type with
represents a message with
+ Message
+ PacketIdentifier
+ Crypto

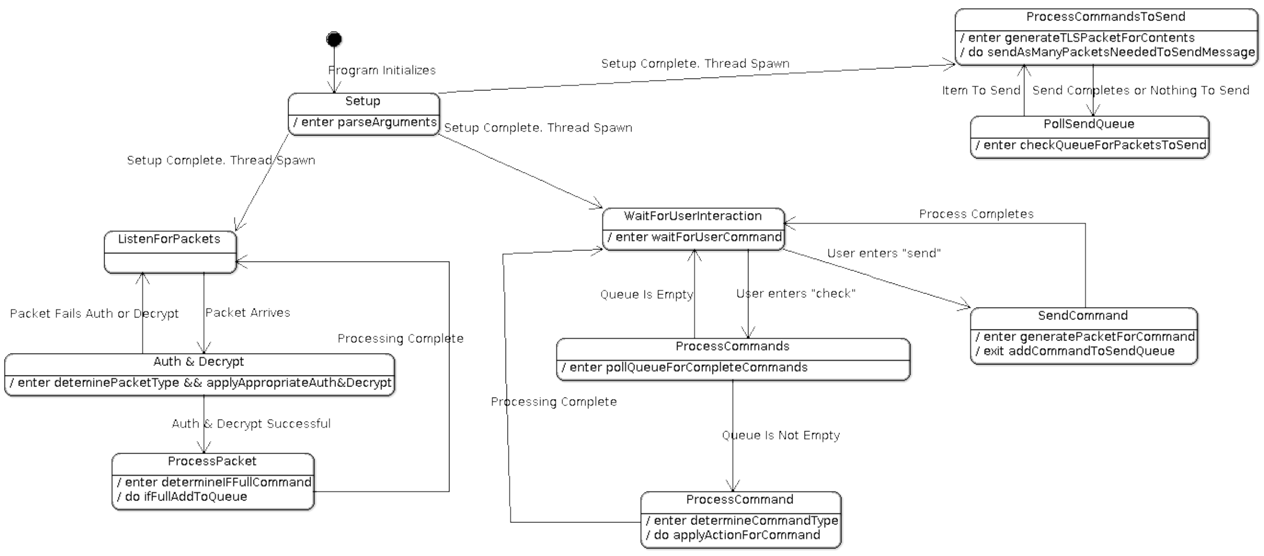
Statechart Diagrams

Haanrad (ExFiltration System) Class Diagram



Program Initializes
Startup
/ enter analyseSystem && listenForDNSRequests
/ do setPasswordToDNSRequest
/ exit sendAuthenticicationPacketToClient
AuthSent
MonitorNetwork
/ enter ListenToNetworkBasedOnAnalysis
Analysis Complete
AnalyzeSystem
/ enter checkSystemRelativeToProcess
/ exit set SystemTimer preferences
Invalid Packet
Packet Arrives
Ready To Listen To Network Again
Are Files To Listen For
ListenForFileEvents
Event Occurs
Decrypt & Authenticate
ConfigureFileEvents
HandleEvent
/ enter handleBasedOnType
Valid Packet
Command Is File Event
Configuration Completes
Ready To Send Response
Handle Command
/ enter executeCommand || createListenerForFileEvents
Command Is Shell Command
Command Completes
SystemTimer
/ enter determineRequestSource && setTimer
/ do waitTilTimerExpires
Ready To Listen For File Events Again
SendResponse
/ enter enforceSingleAccessToSend
Timer Expires

Client Class Diagram



ProcessCommandsToSend
/ enter generateTLSPacketForContents
/ do sendAsManyPacketsNeededToSendMessage
Setup Complete. Thread Spawn
Program Initializes
Item To Send
Send Completes or Nothing To Send
Setup
/ enter parseArguments
Setup Complete. Thread Spawn
PollSendQueue
/ enter checkQueueForPacketsToSend
Setup Complete. Thread Spawn
WaitForUserInteraction
/ enter waitForUserCommand
Process Completes
ListenForPackets
User enters "send"
Queue Is Empty
User enters "check"
Packet Fails Auth or Decrypt
Packet Arrives
Processing Complete
SendCommand
/ enter generatePacketForCommand
/ exit addCommandToSendQueue
Auth & Decrypt
/ enter deteminePacketType && applyAppropriateAuth&Decrypt
ProcessCommands
/ enter pollQueueForCompleteCommands
Processing Complete
Auth & Decrypt Successful
Queue Is Not Empty
ProcessPacket
/ enter determineIFFullCommand
/ do ifFullAddToQueue
ProcessCommand
/ enter determineCommandType
/ do applyActionForCommand

Pesudocode

Haanrad

Main.cpp

*main(int argc, char * argv[])*

```
1. Start in STARTUP mode. Create ProcessDistorter, TrafficAnalyzer, SystemState, Time, SystemState and NetworkMonitor objects.
2. Hide process immediately with ProcessDistorter
3. Configure TrafficAnalyser with new time segment
4. Start NetworkMonitor to listen only for DNS Request traffic (it will know this because SystemState first starts in STARTUP mode, which NetworkMonitor uses to determine what to listen for)
5. Once DNS Request acquired, parse out first query in packet → this will be our password.
6. Set This password for HCrypto and Authenticator objects. Tell SystemState to make evaluation of system.
7. We are now in FULL FUNCTIONALITY mode. Create ExecutorQueue. Create NetworkMonitorThread with NetworkMonitorQueue, spawn these objects on a new thread. Create CovertSocketThread with CovertSocketQueue, spawn these objects on a new thread. Create FileSystemManager, FileSystemManagerThread and FileSystemManagerQueue and spawn these objects on a new thread.
8. Create the Executor object
9. WHILE
    1. Execute ProcessDistorter to evaluate new best name for system
    2. Check with Time for next tick to operate
    3. Execute SystemState to evaluate the system state again
    4. Check with Time for next tick to operate
    5. Check the ExecutorQueue for new Executions. IF there are, execute them with Executor. ELSE continue
```

parseOutDNSQuery(PacketMeta meta)

```
1. Take packet in meta.packet and sift through contents to find the first Query in the packet
2. reformat to a string using human readable ASCII characters
```

3. return the string OR return empty string if cannot parse the packet

ProcessDistorter.cpp

setProcessName(string name)

1. Set this process name to the passed in string name
2. If appendHint is true, append “-bd” to the name of the process. This is for debugging or demos

determineProcessName()

1. Call findPopularProcessName.
2. Error Check. IF returned is nullptr, then determinging failed. IF useDefaultOnFail is true set the process name to the default (kworker/4). ELSE continue using the current name again

findPopularProcessName()

1. Sort through /proc folder to get a list of all processes.
2. Take tally of all process names in /proc/{pid}/comm , duplicates get more points
3. Sort through tally to find most popular, IF there is a draw, rand() 0 or 1 to pick one or the other

SystemState.cpp

SystemState is a wrapper handler that will get system information about the computers current workings. This will answer how much RAN is being used, CPU usage, process counts, and how much work our process is taking

This class should ultimately dictate whether the backdoor should completely go dormant because of lack of system activity or whether it can “gun it out” because there is a lot happening on it

This class should be able to send events that will cause haanrad to slow down to a complete stop

getOutboundBitrate(bool setHistoryOnly)

1. Calculate the outbound bitrate using the ProcHelper class and parsing the /proc/net/dev file
2. IF setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

getInboundBitrate(bool setHistoryOnly)

1. Calculate the inbound bitrate using the ProcHelper class and parsing the /proc/net/dev file
2. IF setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

getAverageProcessCPUUsage(bool setHistoryOnly)

1. Calculate the average CPU usage used by all processes since the last calculation using the ProcHelper and ProcStat classes.
2. IF setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

getPercentageOfCPUUsed(bool setHistoryOnly)

1. Calculate what percent (0 – 100 %) of the CPU the haanrad process has used compared to all other processes on the system since the last calculation using the ProcHelper and ProcStat classes.
2. If setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

getAverageProcessRAMUsage(bool setHistoryOnly)

1. Calculate the average RAM usage used by all processes since the last calculation using the ProcHelper and ProcStat classes.
2. IF setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

getPercentageOfRAMUsed(bool setHistoryOnly)

1. Calculate what percent (0 – 100 %) of the RAM haanrad process has used compared to all other processes on the system since the last calculation using the ProcHelper and ProcStat classes.
2. If setHistoryOnly is true, make the calculations and set the history information needed for the next calculation, don’t actually return the calculation

makeEvaluationOfSystem()

1. Using the above methods, determine the state of Haanrad and set the currentState to a SystemStateMode value. If Haanrad is processing too high, lower the state and increase tick duration in the Time object.

TrafficAnalyzer.cpp

TrafficAnalyzer stores statistic information and packet information on traffic sent to it. Packets are added to its history, which it then evaluates when determining the best packet to send. TrafficAnalyzer is used by the CovertSocket class to replicate packets in the network when communicating back to the client

addPacketMetaToHistory(PacketMeta packet)

1. Add the PacketMeta object (representing the packet and basic information about it) to the deque

getBestPacketToSend()

1. Take a snapshot of the deque currently as it will be updated by other threads during evaluation
2. Using snapshot tally all different kinds of supported packets
3. Select most popular packet. Generate random number between range of deque and find a packet of the most popular type
4. Return that PacketMeta object

getLastPacketAdded()

1. Fetch last PacketMeta object from the end of the deque. This is used in the STARTUP mode.

NetworkMonitor.cpp

NetworkMontior listens for all traffic on the network. It determines whether a packet belongs to Haanrad or not based on the Authenticator and HCrypto objects. If the packet does not belong to Haanrad, it is passed to the TrafficAnalyzer as a sample packet

NetworkMonitor uses the SystemState::currentState static variable to determine what it should listen for.

listenForTraffic()

1. Initialize libpcap, filter and components and start the pcap_loop
2. This method will hang until a valid command has been parsed out of the received network packets from the client

*packetCallback(u_char *ptrnull, const struct pcap_pkthdr * pkt_info const u_char *packet)*

1. Check SystemState:currentState. IF STARTUP only process DNS request packets. No authentication or encryption is needed
2. ELSE we are in FULL FUNCTIONALITY mode. Determine packet type using PacketIdentifier to generate a PacketMeta object.
3. IF packet is TLS → decrypt first with HCrypto and authenticate with Authenticator
4. IF packet is DNS or UDP or TCP → authenticate first with Authenticator and then decrypt with HCrypto
5. IF any of the above steps fail. The packet is not ours, give to TrafficAnalyzer. ELSE the packet is ours.

6. IF packet is TLS or DNS → pass to parseApplicationContent to get packet contents
7. IF packet is TCP or UDP → pass to parseTransportContent to get packet contents
8. Call isFullCommand. IF true then call killListening. This will cause listenForTraffic to unhang and return the command. ELSE return and continue

isFullCommand()

1. Check the command variable as to whether it is a complete valid Haanrad packet sent from the client.
2. Check first letter is '{'. IF not we have likely picked up garbage. Clear command. Return false
3. IF command length is > 5. Check the command starts with {HAAN. IF NOT Return false
4. IF command length is < 11. It can't be complete. Return false
5. Search the command for HAAN} closing tags. If they exist then this is a valid command. Return True

*parseApplicationContent(PacketMeta * meta, char * applicationLayer)*

1. Call isOwnPacket to check this is not our own packet leaving. IF true, return
2. Check ApplicationType
    1. IF TLS → the body contains the now unencrypted payload. Max of 35 bytes
    2. IF DNS → the id header contains the now unencrypted payload. Max of 2 bytes
3. Append contents to the command variable

*parseTransportContent(PacketMeta * meta)*

1. Call isOwnPacket to check this is not our own packet leaving. IF true, return
2. Check TransportType
    1. IF TCP → the sequence number contains the now unencrypted payload. Max of 2 bytes in the 3rd and 4th bytes of the sequence number
    2. IF UDP → the source port contains the noew unencrypted payload. Max of 1 byte in the 2nd byte of the port number.
3. Append contents to the command variable

*isOwnPacket(PacketMeta * meta)*

1. Check the meta→packet for its destination address. If the destination address is for the client, then assume that this is our own packet as it is unlikely the system Haanrad is running on would be communicating with it

getInterface()

1. Fetch the “any” interface from all libpcap interfaces available on the system. Return false on failure
2. Set the allInterfaces variable to all interfaces that are found, and set interface to the “any” interface. This is so that these resources are properly cleaned up upon Haanrad terminating

FileSystemManager.cpp

updateNotifyEvents(Message message)

1. Check with inverseFileEventCommands if the passed notify message is already configured
2. IF it is already configured, assume it is being removed
    1. Find notify decriptor from the inverseFileEventCommands map
    2. Remove notify with inotify_rm_watch
    3. IF failure, send error back to the client. ELSE remove mappings from inverseFileEventCommands, fileEventCommands and eventTypeMap maps.
3. IF is is not configured, assume it is being added
    1. Add notify with inotify_add_watch for IN_CLOSE_WRITE events
    2. IF failure, send error back to the client. ELSE add message.data => dirfd mapping in the inverseFileEventCommands map. Add dirfd => message.data mapping in the fileEventCommands map and add dirfd => MessageTypeEnum in the eventTypeMap

hangForEvents()

1. Configure select structures. Create timer for 10 seconds. Hang on select passing inotifyfd to listen on for read events.
2. IF timeout, return
3. IF event occurs
    1. Read out events. Validate they are IN_CLOSE_WRITE events.
    2. Fetch MessageType from fileEventCommands for dirfd.
        1. IF MessageType::FILE → send an event message back to the client by creating a packet and adding it to the CovertSocketQueue
        2. IF MessageType::FILESYNC → send an event message back to the client by creating a packet and add it to the CovertSocketQueue. Then read the file contents as binary, generate another packet for it, and send it by adding it to the CovertSocketQueue.

Executor.cpp

formatCommand(string haanradPacket)

1. Parse out of the string the command data and message type from the haanrad packet string. Return the formatted Message object or error on failure to parse

execute(Message message)

1. Determine type of message is being executed
2. IF CMD → call exeucteOnConsole
3. IF FILE or FILESYNC → add message to the FileSystemManagerQueue
4. IF SPCCMD → handle specifically based on type
5. ELSE → Command is unknown print error to logger and stop processing

executeOnConsole(Message message)

1. Parse data for console from message
2. check if command is 'cd'. IF cd use chdir command and change command to 'pwd'
3. spawn new shell with popen and execute command
4. read out results from stdout and stderr
5. return result as string

CovertSocket.cpp

CovertSocket will send all data out of the network, using the TrafficAnalyzer to determine what packet is best suited to send at the time. CovertSocket will use the Time object to determine when to send each packet.

send(string payload)

1. If SystemState::curentState is STARTUP. Then fetch the last packet from the TrafficAnalyzer and, adjust the packet to point to the client and send
2. ELSE call getBestPacketToSend from TrafficAnalyzer and determine returned packet type
    1. WHILE payload still to process
        1. IF TLS →Parse max 35 byte chunk of the full payload and place into packet payload. Add authentication with Authenticator and encrypt with Hcrypt and then send
        2. IF DNS → Parse max 2 byte chunk of the full payload and place into id header of DNS. Encrypt with Hcrypt, Add authentication with Authenticator and then send

3. IF TCP → Parse max 2 byte chunk of the full payload and place into 3rd and 4th byte of seuqence number. Encrypt with Hcrypt, Add authentication with Authenticator and then send
4. IF UDP → Parse max 1 byte chunk of the full payload and place in 2nd byte of source port. Encrypt with HCrypt, add authentication with Authenticator and then send

Client

Main.cpp

*main(int argc, char * argv[])*

1. Setup event handlers for Ctrl+C kill command
2. Create MessageQueue, HCrypto, CommHandler objects. Spawn listenr and senderThreads. Both will use the CommHandler object instance
3. Start in STARTUP mode. Wait for packet for DNS packet from Haanrad.
4. Once packet received we are in FULL FUNCTIONALITY mode. Create LocalFileManager object.
5. WHILE keepRunning
    1. Check MessageQueue for new messages
        1. IF new Message → determine type and execute procedure occrdingly
        2. ELSE prompt user for input. Either ‘check’ or ‘send’ commands

CommHandler.cpp

getInterface()

1. Find the “any” interface from the list of interfaces retrieved from libpcap. Return false on failure
2. Set allInterfaces to all interface retreived and interface to the “any” interface. This is so that the resources are properly deleted upon client termination

*packetCallback(u_char *ptrnull, const struct pcap_pkthdr *pkt_into, const u_char *packet)*

1. Check if Haanrad has connected. IF NOT only search for DNS packets. Upon retrieving DNS packet create Message with InterClientMessageType::CONNECTED and put into MessageQueue to be read by main client thread.
2. IF connected, determine packet type and Decrypt with HCrypt and authenticate with Authenticator as appropriate for the packet type

3. Call isFullCommand. IF true call generateMessageFromCommand() and place message into MessageQueue to be retrieved by the main thread

generateMessageFromCommand(string haanradPacket)

1. Parse apart string packet to generate a Message object. IF there is an error generate error message packet and return it

parseOutDNSQuery(PacketMeta meta)

1. Parse apart DNS query and find first Query question domain. This will be the password to communicate with Haanrad. Set the password in HCrypto and Authenticator

listenForMessages()

1. Setup libpcap and filter to listen for udp or tcp traffic. Call the pcap_loop function causing this method to hang until a full command has been parsed by the CommHandler. Calling this method also resets the command variable storing whatever has been gathered so far.

sendPacket(string payload)

1. Send the passed in command packet to Haanrad. Generate a TLS packet and encrypt the payload into it using HCrypt and authenticate it using the Authenticator.

processMessagesToSend()

1. WHILE continueProcessing
    1. poll the MessageQueue for new messages to send. IF they are not empty
        1. parse the message.rawCommandMessage into max 35 byte chunks and send as TLS packets. Call sendPacket() to send the payload chunk for the packet

*parseApplicationContent(PacketMeta * meta, char * applicationLayer)*

4. Call isOwnPacket to check this is not our own packet leaving. IF true, return
5. Check ApplicationType
    1. IF TLS → the body contains the now unencrypted payload. Max of 35 bytes
    2. IF DNS → the id header contains the now unencrypted payload. Max of 2 bytes
6. Append contents to the command variable

*parseTransportContent(PacketMeta * meta)*

1. Call isOwnPacket to check this is not our own packet leaving. IF true, return
2. Check TransportType

    1. IF TCP → the sequence number contains the now unencrypted payload. Max of 2 bytes in the 3rd and 4th bytes of the sequence number
    2. IF UDP → the source port contains the noew unencrypted payload. Max of 1 byte in the 2nd byte of the port number.
3. Append contents to the command variable

isFullCommand()

1. Check the command variable as to whether it is a complete valid Haanrad packet sent from the client.
2. Check first letter is '{'. IF not we have likely picked up garbage. Clear command. Return false
3. IF command length is > 5. Check the command starts with {HAAN. IF NOT Return false
4. IF command length is < 11. It can't be complete. Return false
5. Search the command for HAAN} closing tags. If they exist then this is a valid command. Return True

getCommandBuffer()

1. Returns the current command buffer value. Used for debugging and management of client

clearCommandBuffer()

1. Clears the command buffer regardless of its state. Used for debugging and management of client

LocalFileManager.cpp

buildOutDirectory(string haanradDir)

1. Parse apart haanradDir to find directory path to file
2. Determine if the directory path exists relative to the sync root directory
3. IF it does not exist, recursively create the folder structure needed
4. IF it does exist, return;

syncFile(string haanradDir, string rawData)

1. Parse haanradDir to generate a full path to the local sync root directory and the remote system directory.
2. Parse the fileName from the haanradDir. Write the rawData contents to that file and place it in the parsed directory of the sync root directory.