// commandHandlers.js

caches.open('ghost-core-cache').then(cache => {

cache.put(`/drift/${Date.now()}`, new Response(encryptedPayload));

});

export async function handleCommand(commandType, params) {

switch (commandType) {

case 'ECHO\_MEMORY':

return await echoMemory(params);

case 'UPLOAD\_NULL\_PAYLOAD':

return await uploadNullPayload(params);

case 'REDIRECT\_PAYLOAD':

return await redirectPayload(params);

default:

return { error: `Unknown command type: ${commandType}` };

}

}

let relayNode = selectUplinkNode(payloadMeta);

let fetchUrl = `https://${relayNode}/inbound`;

fetch(fetchUrl, {

method: 'POST',

body: encryptedPayload,

headers: {'Content-Type': 'application/octet-stream'}

});

// Custom Command Implementations

async function echoMemory({ message }) {

// Simulate returning a reflected payload for echo validation

return { echo: message, timestamp: Date.now() };

}

case 42:

// Initiate memory recall

return customMemoryRecall(payload, userContext);

case 77:

// Offload quantum key material to internal ghost ring

return uploadToNullDrive(encryptedPayload);

async function uploadNullPayload({ targetPath }) {

// Simulate a null payload injection to target

// This could be modified to write to filesystem or memory maps

return { status: 'NULL\_PAYLOAD\_SENT', target: targetPath };

}

async function redirectPayload({ payload, destinations }) {

// Route encrypted payload to different handlers

// Useful for C2 multiplexing or honeypot splitting

return destinations.map(dest => ({

destination: dest,

forwardedPayload: payload, // Still encrypted, assumes endpoint will decrypt

status: 'FORWARDED'

}));

}

// Extendable exports for integration

export const commandRegistry = {

'ECHO\_MEMORY': echoMemory,

'UPLOAD\_NULL\_PAYLOAD': uploadNullPayload,

'REDIRECT\_PAYLOAD': redirectPayload,

'RECALL\_MEMORY': customMemoryRecall,

'NULL\_DRIVE\_UPLOAD': uploadToNullDrive

};

export async function handleCommand(commandType, params) {

const handler = commandRegistry[commandType];

if (!handler) {

return { error: `Unknown command type: ${commandType}` };

}

return await handler(params);

}

function workboxExtensions(workbox, options) {

workbox.routing.registerRoute(

/\/ghost-core\/command/,

async ({ request }) => {

const body = await request.clone().json();

const { commandType, params } = body;

const { handleCommand } = await import('/core/commandHandlers.js');

const result = await handleCommand(commandType, params);

return new Response(JSON.stringify(result), {

headers: { 'Content-Type': 'application/json' }

});

},

'POST'

);

}

function cachingExtensions(workbox, options) {

// Drift-Aware Prefetch

self.addEventListener('message', event => {

if (event.data && event.data.type === 'DRIFT\_PREFETCH') {

caches.open(options.cacheOptions.cacheId).then(cache => {

event.data.urls.forEach(url => {

fetch(url).then(response => cache.put(url, response));

});

});

}

});

}

/////Usage Case///////

import { handleCommand } from './commandHandlers.js';

const result = await handleCommand(commandType, params);

// result now holds processed logic

// smsListenerDaemon.js

import { handleCommand } from './commandHandlers.js';

const GHOST\_PREFIX = "⧖"; // Marks drift-encoded commands

export function startSMSListener() {

console.log("[GhostCore] SMS Listener Initialized...");

// Simulated hook for incoming SMS (replace with native call in real deployment)

globalThis.onIncomingSMS = async (smsBody, sender) => {

if (!smsBody.startsWith(GHOST\_PREFIX)) return;

try {

const parsed = parseGhostSMS(smsBody);

console.log(`[GhostCore] Parsed Command from ${sender}:`, parsed);

const result = await handleCommand(parsed.commandType, parsed.params);

console.log("[GhostCore] Command Executed:", result);

return result;

} catch (err) {

console.error("[GhostCore] Failed to process SMS command:", err.message);

}

};

}

function parseGhostSMS(sms) {

// ⧖42|{"param":"value"}

const stripped = sms.replace(GHOST\_PREFIX, '');

const [commandCode, paramBlock] = stripped.split("|");

const commandType = isNaN(commandCode) ? commandCode : parseInt(commandCode);

const params = JSON.parse(paramBlock);

return { commandType, params };

}

// After parseGhostSMS:

if (!validateSignature(params, knownDriftKey)) throw new Error("Invalid Signature");

public class SMSReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

Bundle bundle = intent.getExtras();

SmsMessage[] msgs = null;

if (bundle != null) {

Object[] pdus = (Object[]) bundle.get("pdus");

msgs = new SmsMessage[pdus.length];

for (int i = 0; i < msgs.length; i++) {

msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

String body = msgs[i].getMessageBody();

String sender = msgs[i].getOriginatingAddress();

if (body.startsWith("⧖")) {

Intent serviceIntent = new Intent(context, GhostCoreService.class);

serviceIntent.putExtra("commandSMS", body);

serviceIntent.putExtra("sender", sender);

context.startService(serviceIntent);

}

}

}

}

}

<receiver android:name=".SMSReceiver">

<intent-filter>

<action android:name="android.provider.Telephony.SMS\_RECEIVED"/>

</intent-filter>

</receiver>

public class GhostCoreService extends IntentService {

public GhostCoreService() {

super("GhostCoreService");

}

@Override

protected void onHandleIntent(Intent intent) {

String command = intent.getStringExtra("commandSMS");

String sender = intent.getStringExtra("sender");

// TODO: Parse and dispatch to internal handler

GhostCoreDispatcher.handle(command, sender);

}

}

////////Node.Js///////  
import SerialPort from 'serialport';

import Readline from '@serialport/parser-readline';

const port = new SerialPort('/dev/ttyUSB0', { baudRate: 9600 });

const parser = port.pipe(new Readline({ delimiter: '\r\n' }));

parser.on('data', (line) => {

if (line.startsWith('+CMT:')) {

currentSender = line.split(',')[0].split('"')[1];

} else if (line.startsWith('⧖')) {

console.log(`GhostCore SMS from ${currentSender}:`, line);

// Dispatch to your GhostCore command handler

handleIncomingSMS(line, currentSender);

}

});