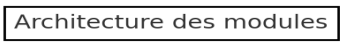
RagnarNet OS — ULTRA-DETAILED Technical Dossier (EN)

Build 2025-08-15 16:12

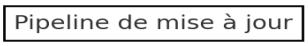
Overview & goals

This document describes RagnarNet OS end-to-end: architecture, flows, security, configuration, database and code. Each module is explained line-by-line with security annotations, plus 12 explanatory diagrams.



Couches de sécurité

Pipeline de mise à jour



Flux d'authentification

Intégrité des fichiers

Intégrité des fichiers

Modèle de privilèges

Modèle de privilèges

Gestion des erreurs

Gestion des erreurs

Flux de configuration

Navigation UI

Schéma users.db

Séquence d’update

Module: update.lua

Detected role: Installer / updater, File operations, Terminal/UI, System reboot, Script execution

Declared functions	BXOR, download, extractCodeVer, fileHash, fnv1a, println, readAll, w
Dependencies (require/shell.run/dofile)	pastebin get

API	Methods used
fs	delete, exists, isDir, open
os	reboot
http	—
shell	run
term	setTextColor

Code & line■by■line explanation

Line 1:
`-- update.lua : Installation / MAJ complète RagnarNet (met à jour manifest & version)`
→ Developer comment.

Line 3: `local function println(c, msg)`
→ Generic Lua instruction.

Line 4: `if term and colors and c then term.setTextColor(c) end`
→ Terminal display interaction (UI).

Line 5: `print(msg)`
→ Generic Lua instruction.

Line 6: `if term and colors then term.setTextColor(colors.white) end`
→ Terminal display interaction (UI).

Line 7: `end`
→ End of block (function/condition/loop).

Line 9: `-- Télécharge via pastebin`
→ Developer comment.

Line 10: `local function download(id, dest)`
→ Generic Lua instruction.

Line 11: `if fs.exists(dest) then fs.delete(dest) end`
→ File deletion (danger: data loss).
■■ File deletion — protect via confirmation/whitelist.

Line 12: `return shell.run("pastebin get " .. id .. " " .. dest)`
→ Executes another script via the shell.
■■ Script execution — avoid untrusted inputs.

Line 13: `end`
→ End of block (function/condition/loop).

Line 15: `local function readAll(p)`
→ Generic Lua instruction.

Line 16: `if not fs.exists(p) or fs.isDir(p) then return "" end`
→ Generic Lua instruction.

Line 17: `local f = fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s`
→ Declares local variable “f” and initializes it to “fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s”.

Line 18: `end`
→ End of block (function/condition/loop).

Line 20: `-- BXOR portable + FNV1a`
→ Developer comment.

Line 21: `local function BXOR(a, b)`
→ Generic Lua instruction.

Line 22: `if bit and bit.bxor then return bit.bxor(a, b) end`
→ Generic Lua instruction.

Line 23: `if bit32 and bit32.bxor then return bit32.bxor(a, b) end`
→ Generic Lua instruction.

Line 24: `local r, v = 0, 1`
→ Generic Lua instruction.

Line 25: `while a > 0 or b > 0 do`
→ WHILE loop: repeats while “a > 0 or b > 0” is true.

Line 26: `local A, B = a % 2, b % 2`
→ Generic Lua instruction.

Line 27: `if (A + B) % 2 == 1 then r = r + v end`
→ Generic Lua instruction.

Line 28: `a = math.floor(a / 2); b = math.floor(b / 2); v = v * 2`
→ Assigns “math.floor(a / 2); b = math.floor(b / 2); v = v * 2” to global variable “a”.

Line 29: `end`
→ End of block (function/condition/loop).

Line 30: `return r`
→ Returns a value to the caller “ r”.

Line 31: `end`
→ End of block (function/condition/loop).

Line 32: `local function fnv1a(s)`
→ Generic Lua instruction.

Line 33: `local h = 2166136261`
→ Declares local variable “h” and initializes it to “2166136261”.

Line 34: `for i = 1, #s do`
→ FOR loop: iterates over “i = 1, #s”.

Line 35: `h = BXOR(h, s:byte(i))`
→ Assigns “BXOR(h, s:byte(i))” to global variable “h”.

Line 36: `h = (h * 16777619) % 4294967296`
→ Assigns “(h * 16777619) % 4294967296” to global variable “h”.

Line 37: `end`
→ End of block (function/condition/loop).

Line 38: `return tostring(h)`
→ Returns a value to the caller “ tostring(h)”.

Line 39: `end`
→ End of block (function/condition/loop).

Line 40: `local function fileHash(path) return fnv1a(readAll(path)) end`
→ Generic Lua instruction.

Line 41:
`local function extractCodeVer(txt) return txt:match('CODE_VER%s*=%s*"%s*([^"]-)%s*"') end`
→ Generic Lua instruction.

Line 43: `-- IDs OFFICIELS (on n'utilise plus ceux de config.lua)`
→ Developer comment.

Line 44: `local files = {`
→ Declares local variable “files” and initializes it to “{”.

Line 45: `{ id = "m7wpD8wF", name = "startup.lua" },`
→ Generic Lua instruction.

Line 46: `{ id = "DWHJU4bC", name = "ui.lua" },`
→ Generic Lua instruction.

Line 47: `{ id = "jK7srvyY", name = "config.lua" },`
→ Generic Lua instruction.

Line 48: `{ id = "gNHAVd7D", name = "update.lua" },`
→ Generic Lua instruction.

Line 49: `}`
→ Generic Lua instruction.

Line 51: `println(colors.cyan, "=== RagnarNet Installer ===")`
→ Generic Lua instruction.

Line 53: `-- 1) Téléchargements`
→ Developer comment.

Line 54: `for _, f in ipairs(files) do`
→ FOR loop: iterates over “_, f in ipairs(files)”.

Line 55: `println(colors.lightBlue, "Telechargement de "..f.name.." ...")`
→ Generic Lua instruction.

Line 56: `local ok = download(f.id, f.name)`
→ Declares local variable “ok” and initializes it to “download(f.id, f.name)”.

Line 57:
```
  if not ok then println(colors.red, "Echec de telechargement: "..f.name); return end
```
→ Generic Lua instruction.

Line 58: `end`
→ End of block (function/condition/loop).

Line 60: `-- 2) Heuristique anti-sabotage pour le startup`
→ Developer comment.

Line 61: `local sTxt = readAll("startup.lua")`
→ Declares local variable “sTxt” and initializes it to “readAll("startup.lua")”.

Line 62: `local ok_ver  = sTxt:match('local%s+CODE_VER%s*=%s*"7%.1%.0"')`
→ Declares local variable “ok_ver” and initializes it to “sTxt:match('local%s+CODE_VER%s*=%s*"7%.1%.0"')”.

Line 63: `local ok_db   = sTxt:match('usersDB%s*=%s*"users%.db"')`
→ Declares local variable “ok_db” and initializes it to “sTxt:match('usersDB%s*=%s*"users%.db"')”.

Line 64: `if not (ok_ver and ok_db) then`
→ IF condition: executes the block if “not (ok_ver and ok_db)” is true.

Line 65: `println(colors.red, "Startup invalide (signature heuristique). Annulation.")`
→ Generic Lua instruction.

Line 66: `return`
→ Returns a value to the caller “”.

Line 67: `end`
→ End of block (function/condition/loop).

Line 69: `-- 3) Manifest & expected version`
→ Developer comment.

Line 70: `local cfg = {}`
→ Declares local variable “cfg” and initializes it to “{}”.

Line 71: `local ver  = extractCodeVer(sTxt) or "7.1.0"`
→ Declares local variable “ver” and initializes it to “extractCodeVer(sTxt) or "7.1.0"”.

Line 73: `cfg.expectedStartupVersion = ver`
→ Generic Lua instruction.

Line 74: `cfg.autoSeal = true`
→ Generic Lua instruction.

Line 75: `cfg.tamperAction, cfg.outdatedAction = "error","error"`
→ Generic Lua instruction.

Line 76: `cfg.askUpdateAtBoot = true`
→ Generic Lua instruction.

Line 77: `cfg.key="RAGNAR123456789KEYULTRA2025"; cfg.protocol="ragnarnet"`
→ Generic Lua instruction.

Line 78: `cfg.adminUser="ragnar"; cfg.adminCode="2013.2013"`
→ Generic Lua instruction.

Line 79: `cfg.spamLimit=5; cfg.maxMessageLength=200; cfg.spamResetTime=300`
→ Generic Lua instruction.

Line 80: `cfg.pepper="RAG-PEPPER-2025"; cfg.pwdHashRounds=512`
→ Generic Lua instruction.

Line 81: `cfg.updateURL_startup="m7wpD8wF"; cfg.updateURL_ui="DWHJU4bC"`
→ Generic Lua instruction.

Line 82: `cfg.updateURL_config="jK7srvyY"; cfg.updateURL_update="gNHAVd7D"`
→ Generic Lua instruction.

Line 83: `cfg.errorCodeTamper=163; cfg.errorCodeOutdated=279`
→ Generic Lua instruction.

Line 84: `cfg.manifest = {`
→ Generic Lua instruction.

Line 85: `["startup.lua"] = fileHash("startup.lua"),`
→ Generic Lua instruction.

Line 86: `["ui.lua"] = fileHash("ui.lua"),`
→ Generic Lua instruction.

Line 87: `["update.lua"] = fileHash("update.lua"),`
→ Generic Lua instruction.

Line 88: `}`
→ Generic Lua instruction.

Line 89: `local function writeConfigTable(tbl)`
→ Generic Lua instruction.

Line 90: `local ser = textutils.serialize(tbl)`
→ Declares local variable “ser” and initializes it to “textutils.serialize(tbl)”.

Line 91: `local f = fs.open("config.lua", "w"); f.write("return " .. ser); f.close()`
→ Declares local variable “f” and initializes it to “fs.open("config.lua", "w"); f.write("return " .. ser); f.close()”.
■■ Write — validate paths & permissions.

Line 92: `end`
→ End of block (function/condition/loop).

Line 93: `writeConfigTable(cfg)`
→ Generic Lua instruction.

Line 95: `println(colors.lime, "Installation terminee. Redemarrage dans 3 secondes...")`
→ Generic Lua instruction.

Line 96: `sleep(3)`
→ Generic Lua instruction.

Line 97: `os.reboot()`
→ Security/maintenance: triggers a system reboot.
■■ Critical action — limit to safe/logged cases.

Module: startup.lua

Detected role: System entry point, File operations, Terminal/UI, Script execution

Declared functions	BXOR, addMessage, askUpdateAtBoot, fileHash, fnv1a, fnvRounds,
Dependencies (require/shell.run/dofile)	config, update.lua

API	Methods used
fs	exists, isDir, open
os	clock, epoch, pullEvent, time
http	—
shell	run
term	clear, getSize, setBackgroundColor, setCursorPos, setTextColor

Code & line■by■line explanation

Line 1: `-- startup.lua : RagnarNet OS principal v7.1.0 (restauré + durci)`
→ Developer comment.

Line 3: `----------------------------------------------------------------`
→ Developer comment.

Line 4: `-- >>> SECURE PREAMBLE (sans hash, anti-suppression d'appel) <<<`
→ Developer comment.

Line 5: `----------------------------------------------------------------`
→ Developer comment.

Line 6: `do`
→ Generic Lua instruction.

Line 7: `-- 1) Fichiers essentiels présents + non vides + marqueurs de structure`
→ Developer comment.

Line 8: `local essentiels = {`
→ Declares local variable “essentiels” and initializes it to “{”.

Line 9: `{ "ui.lua", "return", "drawUI" }, -- doit être un module qui 'return' un tableau + avoir drawUI`
→ Generic Lua instruction.

Line 10: `{ "config.lua", "return", "{" }, -- doit 'return {'`
→ Generic Lua instruction.

Line 11: `{ "users.db", nil, nil }, -- peut être vide au 1er boot, mais doit exister`
→ Generic Lua instruction.

Line 12: `}`
→ Generic Lua instruction.

Line 14: `local function readAll(p)`
→ Generic Lua instruction.

Line 15:
```
    if not fs or not fs.exists or not fs.exists(p) or fs.isDir(p) then return "" end
```
→ Generic Lua instruction.

Line 16: `local f = fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s`
→ Declares local variable “f” and initializes it to “fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s”.

Line 17: `end`
→ End of block (function/condition/loop).

Line 19: `for _, spec in ipairs(essentiels) do`
→ FOR loop: iterates over “_, spec in ipairs(essentiels)”.

Line 20: `local path, m1, m2 = spec[1], spec[2], spec[3]`
→ Generic Lua instruction.

Line 21: `if not fs or not fs.exists or not fs.exists(path) then`
→ IF condition: executes the block if “not fs or not fs.exists or not fs.exists(path)” is true.

Line 22: `error("[SECURITE] Fichier essentiel manquant : "..tostring(path))`
→ Generic Lua instruction.

Line 23: `end`
→ End of block (function/condition/loop).

Line 24: `local data = readAll(path)`
→ Declares local variable “data” and initializes it to “readAll(path)”.

Line 25: `if #data == 0 and path ~= "users.db" then`
→ IF condition: executes the block if “#data == 0 and path ~= "users.db"” is true.

Line 26: `error("[SECURITE] Fichier essentiel vide : "..tostring(path))`
→ Generic Lua instruction.

Line 27: `end`
→ End of block (function/condition/loop).

Line 28: `if m1 and not data:find(m1, 1, true) then`
→ IF condition: executes the block if “m1 and not data:find(m1, 1, true)” is true.

Line 29:
```
error("[SECURITE] Structure invalide dans "..path.." (marqueur "..m1.." absent)")
```
→ Generic Lua instruction.

Line 30: `end`
→ End of block (function/condition/loop).

Line 31: `if m2 and not data:find(m2, 1, true) then`
→ IF condition: executes the block if “m2 and not data:find(m2, 1, true)” is true.

Line 32:
```
error("[SECURITE] Structure invalide dans "..path.." (marqueur "..m2.." absent)")
```
→ Generic Lua instruction.

Line 33: `end`
→ End of block (function/condition/loop).

Line 34: `end`
→ End of block (function/condition/loop).

Line 35: `end`
→ End of block (function/condition/loop).

Line 36: `-- >>> FIN PREAMBLE <<<`
→ Developer comment.

Line 38: `local CODE_VER = "7.1.0"`
→ Declares local variable “CODE_VER” and initializes it to “"7.1.0"”.

Line 39: `local cfg = require("config")`
→ Declares local variable “cfg” and initializes it to “require("config")”.

Line 41: `------------------------------`
→ Developer comment.

Line 42: `-- Garde-fous config forcés`
→ Developer comment.

Line 43: `------------------------------`
→ Developer comment.

Line 44: `cfg.spamLimit        = math.max(1,  math.min(50,   tonumber(cfg.spamLimit or 5)))`
→ Generic Lua instruction.

Line 45:
```
cfg.maxMessageLength = math.max(10, math.min(500,  tonumber(cfg.maxMessageLength or 200)))
```
→ Generic Lua instruction.

Line 46:
```
cfg.pwdHashRounds    = math.max(128,math.min(4096, tonumber(cfg.pwdHashRounds or 512)))
```
→ Generic Lua instruction.

Line 47: `if cfg.strict_mode == false then cfg.strict_mode = true end`
→ Generic Lua instruction.

Line 48: `-- Empêcher un contournement via actions trop ?douces?`
→ Developer comment.

Line 49:
```
cfg.tamperAction   = (cfg.tamperAction   == "halt" or cfg.tamperAction   == "erro
r") and cfg.tamperAction   or "error"
```
→ Generic Lua instruction.

Line 50:
```
cfg.outdatedAction = (cfg.outdatedAction == "halt" or cfg.outdatedAction == "erro
r") and cfg.outdatedAction or "error"
```
→ Generic Lua instruction.

Line 52: `----------------`
→ Developer comment.

Line 53: `-- Utils`
→ Developer comment.

Line 54: `----------------`
→ Developer comment.

Line 55: `local function BXOR(a, b)`
→ Generic Lua instruction.

Line 56: `  if bit and bit.bxor then return bit.bxor(a, b) end`
→ Generic Lua instruction.

Line 57: `  if bit32 and bit32.bxor then return bit32.bxor(a, b) end`
→ Generic Lua instruction.

Line 58: `  local r, v = 0, 1`
→ Generic Lua instruction.

Line 59: `  while a > 0 or b > 0 do`
→ WHILE loop: repeats while “a > 0 or b > 0” is true.

Line 60: `    local A, B = a % 2, b % 2`
→ Generic Lua instruction.

Line 61: `    if (A + B) % 2 == 1 then r = r + v end`
→ Generic Lua instruction.

Line 62: `    a = math.floor(a / 2); b = math.floor(b / 2); v = v * 2`
→ Assigns “math.floor(a / 2); b = math.floor(b / 2); v = v * 2” to global variable “a”.

Line 63: `  end`
→ End of block (function/condition/loop).

Line 64: `  return r`
→ Returns a value to the caller “ r”.

Line 65: `end`
→ End of block (function/condition/loop).

Line 67: `local function readAll(p)`
→ Generic Lua instruction.

Line 68: `  if not fs.exists(p) or fs.isDir(p) then return "" end`
→ Generic Lua instruction.

Line 69: `  local f = fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s`
→ Declares local variable “f” and initializes it to “fs.open(p, "r"); local s = f.readAll() or ""; f.close(); return s”.

Line 70: `end`
→ End of block (function/condition/loop).

Line 72: `local function fnv1a(s)`
→ Generic Lua instruction.

Line 73: `  local h = 2166136261`
→ Declares local variable “h” and initializes it to “2166136261”.

Line 74: `  for i = 1, #s do`
→ FOR loop: iterates over “i = 1, #s”.

Line 75: `    h = BXOR(h, s:byte(i))`
→ Assigns “BXOR(h, s:byte(i))” to global variable “h”.

Line 76: `    h = (h * 16777619) % 4294967296`
→ Assigns “(h * 16777619) % 4294967296” to global variable “h”.

Line 77: `  end`
→ End of block (function/condition/loop).

Line 78: `  return tostring(h)`
→ Returns a value to the caller “ tostring(h)”.

Line 79: `end`
→ End of block (function/condition/loop).

Line 80: `local function fileHash(path) return fnv1a(readAll(path)) end`
→ Generic Lua instruction.

Line 82: `local function writeConfigTable(tbl)`
→ Generic Lua instruction.

Line 83: `  local ser = textutils.serialize(tbl)`
→ Declares local variable “ser” and initializes it to “textutils.serialize(tbl)”.

Line 84: `local f = fs.open("config.lua", "w"); f.write("return " .. ser); f.close()`
→ Declares local variable “f” and initializes it to “fs.open("config.lua", "w"); f.write("return " .. ser); f.close()”.
■■ Write — validate paths & permissions.

Line 85: `end`
→ End of block (function/condition/loop).

Line 87: `------------------------`
→ Developer comment.

Line 88: `-- Erreurs propres`
→ Developer comment.

Line 89: `------------------------`
→ Developer comment.

Line 90: `local function showErrorAndExit(code, reason)`
→ Generic Lua instruction.

Line 91: `term.setBackgroundColor(colors.black)`
→ Terminal display interaction (UI).

Line 92: `term.setTextColor(colors.red)`
→ Terminal display interaction (UI).

Line 93: `term.clear()`
→ Terminal display interaction (UI).

Line 94: `term.setCursorPos(2,2)`
→ Terminal display interaction (UI).

Line 95: `print("[ERREUR " .. tostring(code) .. "] RagnarNet")`
→ Generic Lua instruction.

Line 96: `term.setTextColor(colors.white)`
→ Terminal display interaction (UI).

Line 97: `print(reason or "Erreur de securite")`
→ Generic Lua instruction.

Line 98:
```
print("\nLe programme s'arrete. Lance 'update' ou utilise la disquette de recovery.")
```
→ Generic Lua instruction.

Line 99: `sleep(2.5)`
→ Generic Lua instruction.

Line 100: `error("ERR_"..tostring(code), 0)`
→ Generic Lua instruction.

Line 101: `end`
→ End of block (function/condition/loop).

Line 103: `local function handleBreach(kind, reason, action)`
→ Generic Lua instruction.

Line 104: `action = action or "error"`
→ Assigns “action or "error"” to global variable “action”.

Line 105: `local code = 199`
→ Declares local variable “code” and initializes it to “199”.

Line 106: `if kind == "tamper"   then code = (cfg.errorCodeTamper or 163) end`
→ Generic Lua instruction.

Line 107: `if kind == "outdated" then code = (cfg.errorCodeOutdated or 279) end`
→ Generic Lua instruction.

Line 108: `if action == "error" then showErrorAndExit(code, reason)`
→ Generic Lua instruction.

Line 109: `elseif action == "halt" then error(reason or "Security halt", 0)`
→ Generic Lua instruction.

Line 110: `else showErrorAndExit(199, reason or "Security error") end`
→ Generic Lua instruction.

Line 111: `end`
→ End of block (function/condition/loop).

Line 113: `------------------------------------------------`
→ Developer comment.

Line 114: `-- Intégrité (avec anti-reseal et auto-run)`
→ Developer comment.

Line 115: `----------------------------------------------------`
→ Developer comment.

Line 116: `local function integrityCheck()`
→ Generic Lua instruction.

Line 117: `-- Anti-reseal : un seul ?seal? autorisé à l?installation`
→ Developer comment.

Line 118: `local sealedFlag = ".sealed"`
→ Declares local variable “sealedFlag” and initializes it to “".sealed"”.

Line 119: `local firstBoot = not fs.exists(sealedFlag)`
→ Declares local variable “firstBoot” and initializes it to “not fs.exists(sealedFlag)”.

Line 121: `-- 1) Version attendue (si définie dans config)`
→ Developer comment.

Line 122: `if cfg.expectedStartupVersion and cfg.expectedStartupVersion ~= CODE_VER then`
→ IF condition: executes the block if “cfg.expectedStartupVersion and cfg.expectedStartupVersion ~= CODE_VER” is true.

Line 123: `handleBreach("outdated",`
→ Generic Lua instruction.

Line 124: `"Version trop ancienne: "..tostring(CODE_VER).." (attendue "..tostring(cfg.expectedStartupVersion)..")",`
→ Generic Lua instruction.

Line 125: `cfg.outdatedAction or cfg.tamperAction or "error")`
→ Generic Lua instruction.

Line 126: `end`
→ End of block (function/condition/loop).

Line 128: `-- 2) Cibles d?intégrité ? on n?inclut jamais config.lua`
→ Developer comment.

Line 129: `local targets = { "startup.lua", "ui.lua", "update.lua" }`
→ Declares local variable “targets” and initializes it to “{ "startup.lua", "ui.lua", "update.lua" }”.

Line 131: `-- 3) Manifest`
→ Developer comment.

Line 132: `if not cfg.manifest then`
→ IF condition: executes the block if “not cfg.manifest” is true.

Line 133: `if cfg.autoSeal and firstBoot then`
→ IF condition: executes the block if “cfg.autoSeal and firstBoot” is true.

Line 134: `local newcfg = {}; for k,v in pairs(cfg) do newcfg[k] = v end`
→ Declares local variable “newcfg” and initializes it to “{}; for k,v in pairs(cfg) do newcfg[k] = v end”.

Line 135: `newcfg.manifest = {}`
→ Generic Lua instruction.

Line 136: `for _, p in ipairs(targets) do newcfg.manifest[p] = fileHash(p) end`
→ Generic Lua instruction.

Line 137: `writeConfigTable(newcfg)`
→ Generic Lua instruction.

Line 138: `local f = fs.open(sealedFlag, "w"); f.write("ok"); f.close()`
→ Declares local variable “f” and initializes it to “fs.open(sealedFlag, "w"); f.write("ok"); f.close()”.
■■ Write — validate paths & permissions.

Line 139: `term.setTextColor(colors.lime); print("[Integrite] Scellage initial OK (manifest ecrit)."); term.setTextColor(colors.white)`
→ Terminal display interaction (UI).

Line 140: `else`
→ ELSE branch: default case when previous conditions are false.

Line 141:
`-- Si manifest absent mais pas ?vraie? installation : tentative de reseal -> breach`
→ Developer comment.

Line 142: `local msg = firstBoot and "Manifest absent et autoSeal=false (installation corrompue)."`
→ Declares local variable “msg” and initializes it to “firstBoot and "Manifest absent et autoSeal=false (installation corrompue).”.

Line 143:
`or "Manifest absent (tentative de re-scellage interdite)."`
→ Generic Lua instruction.

Line 144: `handleBreach("tamper", msg, cfg.tamperAction or "error")`
→ Generic Lua instruction.

Line 145: `end`
→ End of block (function/condition/loop).

Line 146: `else`
→ ELSE branch: default case when previous conditions are false.

Line 147: `for _, p in ipairs(targets) do`
→ FOR loop: iterates over “_, p in ipairs(targets)”.

Line 148: `local exp, act = cfg.manifest[p], fileHash(p)`
→ Generic Lua instruction.

Line 149: `if not exp or exp ~= act then`
→ IF condition: executes the block if “not exp or exp ~= act” is true.

Line 150:
```
handleBreach("tamper", "Integrite rompue sur: "..p, cfg.tamperAction or "error")
```
→ Generic Lua instruction.

Line 151: `end`
→ End of block (function/condition/loop).

Line 152: `end`
→ End of block (function/condition/loop).

Line 153: `end`
→ End of block (function/condition/loop).

Line 154: `end`
→ End of block (function/condition/loop).

Line 156: `-- AUTO-RUN : vérifie l?intégrité même si l?appel plus bas est supprimé`
→ Developer comment.

Line 157: `do`
→ Generic Lua instruction.

Line 158: `local ok, err = pcall(integrityCheck)`
→ Protected call (pcall) to capture errors.

Line 159: `if not ok then`
→ IF condition: executes the block if “not ok” is true.

Line 160:
```
handleBreach("tamper", "Echec verif integrite: "..tostring(err), cfg.tamperA
ction or "error")
```
→ Generic Lua instruction.

Line 161: `end`
→ End of block (function/condition/loop).

Line 162: `end`
→ End of block (function/condition/loop).

Line 164: `----------`
→ Developer comment.

Line 165: `-- MAJ au démarrage (prompt unique)`
→ Developer comment.

Line 166: `----------`
→ Developer comment.

Line 167: `local _askedUpdateOnce = false`
→ Declares local variable “_askedUpdateOnce” and initializes it to “false”.

Line 168: `local function askUpdateAtBoot()`
→ Generic Lua instruction.

Line 169: `if _askedUpdateOnce then return end`
→ Generic Lua instruction.

Line 170: `_askedUpdateOnce = true`
→ Assigns “true” to global variable “_askedUpdateOnce”.

Line 171: `if cfg.askUpdateAtBoot == false then return end`
→ Generic Lua instruction.

Line 172: `if not fs.exists("update.lua") then return end`
→ Generic Lua instruction.

Line 173:
```
term.setTextColor(colors.cyan); print("\nFaire la mise a jour maintenant ? (o/
n)"); term.setTextColor(colors.white)
```
→ Terminal display interaction (UI).

Line 174: `local a = read()`
→ Declares local variable “a” and initializes it to “read()”.
■■ Input — mask passwords, filter injections.

Line 175: `if a and a:lower() == "o" then`
→ IF condition: executes the block if “a and a:lower() == "o"” is true.

Line 176: `local dat = readAll("update.lua")`
→ Declares local variable “dat” and initializes it to “readAll("update.lua")”.

Line 177: `if dat == "" then showErrorAndExit(503, "update.lua invalide ou vide") end`
→ Generic Lua instruction.

Line 178: `shell.run("update.lua")`
→ Executes another script via the shell.
■■ Script execution — avoid untrusted inputs.

Line 179: `end`
→ End of block (function/condition/loop).

Line 180: `end`
→ End of block (function/condition/loop).

Line 182: `-- ====== Boot: intégrité -> prompt MAJ -> charge UI ======`
→ Developer comment.

Line 183: `-- Même si quelqu?un supprime la ligne suivante, l?intégrité a déjà été vérifiée (auto-run ci-dessus).`
→ Developer comment.

Line 184: `integrityCheck()`
→ Generic Lua instruction.

Line 185: `askUpdateAtBoot()`
→ Generic Lua instruction.

Line 187: `---------------------------------`
→ Developer comment.

Line 188: `-- Chargement et vérif de l'UI`
→ Developer comment.

Line 189: `---------------------------------`
→ Developer comment.

Line 190: `local function loadUI()`
→ Generic Lua instruction.

Line 191: `if package and package.loaded then package.loaded["ui"] = nil end`
→ Generic Lua instruction.

Line 192: `local ok, mod = pcall(dofile, "ui.lua")`
→ Protected call (pcall) to capture errors.

Line 193: `if not ok then showErrorAndExit(501, "ui.lua: "..tostring(mod)) end`
→ Generic Lua instruction.

Line 194: `if type(mod) ~= "table" or not mod.drawUI or not mod.showMessages then`
→ IF condition: executes the block if “type(mod) ~= "table" or not mod.drawUI or not mod.showMessages” is true.

Line 195: `showErrorAndExit(502, "ui.lua invalide (fonctions manquantes)")`
→ Generic Lua instruction.

Line 196: `end`
→ End of block (function/condition/loop).

Line 197: `return mod`
→ Returns a value to the caller “ mod”.

Line 198: `end`
→ End of block (function/condition/loop).

Line 199: `local ui = loadUI()`
→ Declares local variable “ui” and initializes it to “loadUI()”.

Line 201: `----------------------------`
→ Developer comment.

Line 202: `-- App / runtime`
→ Developer comment.

Line 203: `----------------------------`
→ Developer comment.

Line 204: `local w, h     = term.getSize()`
→ Terminal display interaction (UI).

Line 205: `local uiHeight = h - 6`
→ Declares local variable “uiHeight” and initializes it to “h - 6”.

Line 206: `local usersDB  = "users.db"`
→ Declares local variable “usersDB” and initializes it to ““users.db””.

Line 208: `-- Nettoyage d?artefacts connus`
→ Developer comment.

Line 209:
`for _, f in ipairs({"HACKER.db"}) do if fs.exists(f) then pcall(fs.delete, f) end end`
→ Protected call (pcall) to capture errors.

Line 211: `local messages, users, spamTracker, blacklist, banDuration = {}, {}, {}, {}, {}`
→ Generic Lua instruction.

Line 212: `local username, isAdmin, lockdown = "?", false, false`
→ Generic Lua instruction.

Line 214: `local function xorCrypt(msg, keyStr)`
→ Generic Lua instruction.

Line 215: `local out = {}`
→ Declares local variable “out” and initializes it to “{}”.

Line 216: `for i = 1, #msg do`
→ FOR loop: iterates over “i = 1, #msg”.

Line 217: `local m = msg:byte(i)`
→ Declares local variable “m” and initializes it to “msg:byte(i)”.

Line 218: `local k = keyStr:byte((i - 1) % #keyStr + 1)`
→ Declares local variable “k” and initializes it to “keyStr:byte((i - 1) % #keyStr + 1)”.

Line 219: `out[i] = string.char(BXOR(m, k))`
→ Generic Lua instruction.

Line 220: `end`
→ End of block (function/condition/loop).

Line 221: `return table.concat(out)`
→ Returns a value to the caller “ table.concat(out)”.

Line 222: `end`
→ End of block (function/condition/loop).

Line 224: `local function addMessage(from, text, adminFlag)`
→ Generic Lua instruction.

Line 225: `table.insert(messages, { id = #messages + 1, from = from, text = text, admin = adminFlag or false })`
→ Generic Lua instruction.

Line 226: `ui.drawUI(username, isAdmin, w, h, CODE_VER)`
→ Generic Lua instruction.

Line 227: `ui.showMessages(messages, uiHeight, blacklist, cfg.adminUser)`
→ Generic Lua instruction.

Line 228: `end`
→ End of block (function/condition/loop).

Line 230: `-- Users (hash + migration)`
→ Developer comment.

Line 231: `local function loadUsers()`
→ Generic Lua instruction.

Line 232: `if not fs.exists(usersDB) then return {} end`
→ Generic Lua instruction.

Line 233:
`local f = fs.open(usersDB, "r"); local d = textutils.unserialize(f.readAll()); f.close()`
→ Declares local variable “f” and initializes it to “fs.open(usersDB, "r"); local d = textutils.unserialize(f.readAll()); f.close()”.

Line 234: `return d or {}`
→ Returns a value to the caller “ d or {}”.

Line 235: `end`
→ End of block (function/condition/loop).

Line 236: `local function saveUsers(u)`
→ Generic Lua instruction.

Line 237: `local f = fs.open(usersDB, "w"); f.write(textutils.serialize(u)); f.close()`
→ Declares local variable “f” and initializes it to “fs.open(usersDB, "w"); f.write(textutils.serialize(u)); f.close()”.
■■ Write — validate paths & permissions.

Line 238: `end`
→ End of block (function/condition/loop).

Line 239: `local function randHex(n) local s={} for i=1,n do s[i]=string.format("%x",math.random(0,15)) end return table.concat(s) end`
→ Generic Lua instruction.

Line 240: `local function fnvRounds(s) local r=tonumber(cfg.pwdHashRounds or 512) or 512; local h=s; for _=1,r do h=fnv1a(h) end; return h end`
→ Generic Lua instruction.

Line 241: `local function hashPassword(pwd, salt) return fnvRounds(tostring(salt or "") .. tostring(pwd or "") .. tostring(cfg.pepper or "")) end`
→ Generic Lua instruction.

Line 242: `local function verifyPassword(stored, input)`
→ Generic Lua instruction.

Line 243: `if type(stored)=="string" then return stored==input, "legacy"`
→ Generic Lua instruction.

Line 244: `elseif type(stored)=="table" and stored.salt and stored.hash then return stored.hash==hashPassword(input,stored.salt),"hashed" end`
→ Generic Lua instruction.

Line 245: `return false,"unknown"`
→ Returns a value to the caller “ false,"unknown"”.

Line 246: `end`
→ End of block (function/condition/loop).

Line 248: `-- Modem`
→ Developer comment.

Line 249: `for _, side in ipairs({"left","right","top","bottom","front","back"}) do`
→ FOR loop: iterates over “_, side in ipairs({"left","right","top","bottom","front","back"})”.

Line 250: `if peripheral.getType(side) == "modem" then rednet.open(side); break end`
→ Generic Lua instruction.

Line 251: `end`
→ End of block (function/condition/loop).

Line 253: `-- Login`
→ Developer comment.

Line 254: `users = loadUsers()`
→ Assigns “loadUsers()” to global variable “users”.

Line 255: `math.randomseed(os.epoch and os.epoch("utc") or os.time() or os.clock())`
→ Generic Lua instruction.

Line 257:
`term.setTextColor(colors.yellow) write("Pseudo > "); term.setTextColor(colors.white)`
→ Terminal display interaction (UI).

Line 258: `username = read()`
→ Assigns “read()” to global variable “username”.
■■ Input — mask passwords, filter injections.

Line 260: `if users[username] then`
→ IF condition: executes the block if “users[username]” is true.

Line 261: `term.setTextColor(colors.yellow) write("Mot de passe > "); term.setTextColor(colors.white)`
→ Terminal display interaction (UI).

Line 262: `local pwd = read("*")`
→ Declares local variable “pwd” and initializes it to “read("*")”.
■■ Input — mask passwords, filter injections.

Line 263: `local ok, mode = verifyPassword(users[username], pwd)`
→ Generic Lua instruction.

Line 264: `if not ok then print("Mot de passe incorrect.") return end`
→ Generic Lua instruction.

Line 265: `if mode == "legacy" then`
→ IF condition: executes the block if “mode == "legacy"” is true.

Line 266: `local salt = randHex(16)`
→ Declares local variable “salt” and initializes it to “randHex(16)”.

Line 267: `users[username] = { salt = salt, hash = hashPassword(pwd, salt) }`
→ Generic Lua instruction.

Line 268: `saveUsers(users)`
→ Generic Lua instruction.

Line 269: `addMessage("SYSTEM", "Compte migre vers hachage.", false)`
→ Generic Lua instruction.

Line 270: `end`
→ End of block (function/condition/loop).

Line 271: `else`
→ ELSE branch: default case when previous conditions are false.

Line 272: `term.setTextColor(colors.yellow) write("Creer un mot de passe > "); term.setTe xtColor(colors.white)`
→ Terminal display interaction (UI).

Line 273: `local pwd = read("*"); local salt = randHex(16)`
→ Declares local variable “pwd” and initializes it to “read("*"); local salt = randHex(16)”.
■■ Input — mask passwords, filter injections.

Line 274: `users[username] = { salt = salt, hash = hashPassword(pwd, salt) }`
→ Generic Lua instruction.

Line 275: `saveUsers(users)`
→ Generic Lua instruction.

Line 276: `end`
→ End of block (function/condition/loop).

Line 278: `if username == cfg.adminUser then`
→ IF condition: executes the block if “username == cfg.adminUser” is true.

Line 279: `write("Code Ragnar > ")`
→ Generic Lua instruction.

Line 280:
```
  if read() == cfg.adminCode then isAdmin = true else print("Code incorrect.") return end
```
→ Generic Lua instruction.
■■ Input — mask passwords, filter injections.

Line 281: `end`
→ End of block (function/condition/loop).

Line 283: `-- UI initiale`
→ Developer comment.

Line 284: `ui.drawUI(username, isAdmin, w, h, CODE_VER)`
→ Generic Lua instruction.

Line 285: `ui.showMessages(messages, uiHeight, blacklist, cfg.adminUser)`
→ Generic Lua instruction.

Line 287: `-- Watchdog runtime : re-vérifie l?essentiel régulièrement`
→ Developer comment.

Line 288: `local function watchdog()`
→ Generic Lua instruction.

Line 289: `while true do`
→ WHILE loop: repeats while “true” is true.

Line 290: `-- re-check fichiers essentiels`
→ Developer comment.

Line 291: `local ok, err = pcall(function()`
→ Protected call (pcall) to capture errors.

Line 292: `local marks = {`
→ Declares local variable “marks” and initializes it to “{”.

Line 293: `{ "ui.lua",     "return", "drawUI" },`
→ Generic Lua instruction.

Line 294: `{ "config.lua", "return", "{"      },`
→ Generic Lua instruction.

Line 295: `{ "users.db",   nil,      nil      },`
→ Generic Lua instruction.

Line 296: `}`
→ Generic Lua instruction.

Line 297: `for _, spec in ipairs(marks) do`
→ FOR loop: iterates over “_, spec in ipairs(marks)”.

Line 298: `local path, m1, m2 = spec[1], spec[2], spec[3]`
→ Generic Lua instruction.

Line 299: `if not fs.exists(path) then error("Essentiel supprimé: "..path) end`
→ Generic Lua instruction.

Line 300: `local data = readAll(path)`
→ Declares local variable “data” and initializes it to “readAll(path)”.

Line 301:
`if #data == 0 and path ~= "users.db" then error("Essentiel vide: "..path) end`
→ Generic Lua instruction.

Line 302: `if m1 and not data:find(m1, 1, true) then error("Structure invalide (man que "..m1..") dans "..path) end`
→ Generic Lua instruction.

Line 303: `if m2 and not data:find(m2, 1, true) then error("Structure invalide (man que "..m2..") dans "..path) end`
→ Generic Lua instruction.

Line 304: `end`
→ End of block (function/condition/loop).

Line 305: `end)`
→ Generic Lua instruction.

Line 306: `if not ok then`
→ IF condition: executes the block if “not ok” is true.

Line 307: `handleBreach("tamper", "Watchdog: "..tostring(err), "error")`
→ Generic Lua instruction.

Line 308: `end`
→ End of block (function/condition/loop).

Line 309: `sleep(math.max(2, tonumber(cfg.watchdogDelay or 5)))`
→ Generic Lua instruction.

Line 310: `end`
→ End of block (function/condition/loop).

Line 311: `end`
→ End of block (function/condition/loop).

Line 313: `-- Threads`
→ Developer comment.

Line 314:
`local function spamReset() while true do sleep(cfg.spamResetTime) spamTracker = {} end end`
→ Generic Lua instruction.

Line 315: `local function handleClick()`
→ Generic Lua instruction.

Line 316: `while true do`
→ WHILE loop: repeats while “true” is true.

Line 317: `local _, _, x, y = os.pullEvent("mouse_click")`
→ Generic Lua instruction.

Line 318: `if y == 1 and x >= w - 12 then`
→ IF condition: executes the block if “y == 1 and x >= w - 12” is true.

Line 319: `term.setTextColor(colors.red) print("\nConfirmer l'arret ? (o/n)")`
→ Terminal display interaction (UI).

Line 320: `term.setTextColor(colors.white)`
→ Terminal display interaction (UI).

Line 321: `if (read() or ""):lower() == "o" then error("Arret utilisateur", 0) end`
→ Generic Lua instruction.
■■ Input — mask passwords, filter injections.

Line 322: `end`
→ End of block (function/condition/loop).

Line 323: `end`
→ End of block (function/condition/loop).

Line 324: `end`
→ End of block (function/condition/loop).

Line 325: `local function receiver()`
→ Generic Lua instruction.

Line 326: `  while true do`
→ WHILE loop: repeats while “true” is true.

Line 327: `    local _, encrypted = rednet.receive(cfg.protocol)`
→ Generic Lua instruction.

Line 328: `    local raw = (encrypted and cfg.key) and (function(m,k)`
→ Declares local variable “raw” and initializes it to “(encrypted and cfg.key) and (function(m,k)”.

Line 329: `      local out, b = {}, nil`
→ Generic Lua instruction.

Line 330: `      for i = 1, #m do`
→ FOR loop: iterates over “i = 1, #m”.

Line 331: `        local mm = m:byte(i)`
→ Declares local variable “mm” and initializes it to “m:byte(i)”.

Line 332: `        local kk = k:byte((i - 1) % #k + 1)`
→ Declares local variable “kk” and initializes it to “k:byte((i - 1) % #k + 1)”.

Line 333: `        out[i] = string.char(BXOR(mm, kk))`
→ Generic Lua instruction.

Line 334: `      end`
→ End of block (function/condition/loop).

Line 335: `      return table.concat(out)`
→ Returns a value to the caller “ table.concat(out)”.

Line 336: `    end)(encrypted, cfg.key) or ""`
→ Generic Lua instruction.

Line 337: `    local from, text = raw:match("(.+):(.+)")`
→ Generic Lua instruction.

Line 338: `    if not from or not text then`
→ IF condition: executes the block if “not from or not text” is true.

Line 339: `      -- DoS fix : on ignore le paquet mal formé`
→ Developer comment.

Line 340: `      sleep(0)`
→ Generic Lua instruction.

Line 341: `    else`
→ ELSE branch: default case when previous conditions are false.

Line 342: `      if #text > cfg.maxMessageLength then`
→ IF condition: executes the block if “#text > cfg.maxMessageLength” is true.

Line 343: `        addMessage("SYSTEM","Message trop long de "..from)`
→ Generic Lua instruction.

Line 344: `      elseif blacklist[from] then`
→ ELSEIF branch: alternative if “blacklist[from]” is true.

Line 345: `        -- ignore`
→ Developer comment.

Line 346: `      elseif from ~= username then`
→ ELSEIF branch: alternative if “from ~= username” is true.

Line 347: `        spamTracker[from] = (spamTracker[from] or 0) + 1`
→ Generic Lua instruction.

Line 348: `        if spamTracker[from] > cfg.spamLimit and from ~= cfg.adminUser then`
→ IF condition: executes the block if “spamTracker[from] > cfg.spamLimit and from ~= cfg.adminUser” is true.

Line 349: `          banDuration[from] = (banDuration[from] or 7200) * 5`
→ Generic Lua instruction.

Line 350: `          blacklist[from] = true`
→ Generic Lua instruction.

Line 351: `          addMessage("SYSTEM", from.." banni pour spam")`
→ Generic Lua instruction.

Line 352: `        else`
→ ELSE branch: default case when previous conditions are false.

Line 353: `addMessage(from, text, (from == cfg.adminUser))`
→ Generic Lua instruction.

Line 354: `end`
→ End of block (function/condition/loop).

Line 355: `end`
→ End of block (function/condition/loop).

Line 356: `end`
→ End of block (function/condition/loop).

Line 357: `end`
→ End of block (function/condition/loop).

Line 358: `end`
→ End of block (function/condition/loop).

Line 359: `local function sender()`
→ Generic Lua instruction.

Line 360: `while true do`
→ WHILE loop: repeats while “true” is true.

Line 361: `term.setCursorPos(2, h); term.setTextColor(colors.white); write("Vous > ")`
→ Terminal display interaction (UI).

Line 362: `local input = read()`
→ Declares local variable “input” and initializes it to “read()”.
■■ Input — mask passwords, filter injections.

Line 363: `if input ~= "" and not lockdown then`
→ IF condition: executes the block if “input ~= "" and not lockdown” is true.

Line 364: `if #input > cfg.maxMessageLength then`
→ IF condition: executes the block if “#input > cfg.maxMessageLength” is true.

Line 365:
```
addMessage("SYSTEM","Message trop long (max "..cfg.maxMessageLength..")")
```
→ Generic Lua instruction.

Line 366: `else`
→ ELSE branch: default case when previous conditions are false.

Line 367: `local plain = username..":"..input`
→ Declares local variable “plain” and initializes it to “username..":"..input”.

Line 368: `rednet.broadcast(xorCrypt(plain, cfg.key), cfg.protocol)`
→ Generic Lua instruction.

Line 369: `addMessage(username, input, isAdmin)`
→ Generic Lua instruction.

Line 370: `end`
→ End of block (function/condition/loop).

Line 371: `end`
→ End of block (function/condition/loop).

Line 372: `end`
→ End of block (function/condition/loop).

Line 373: `end`
→ End of block (function/condition/loop).

Line 375: `parallel.waitForAny(receiver, sender, handleClick, spamReset, watchdog)`
→ Generic Lua instruction.

Module: config.lua

Detected role: Global configuration

Declared functions	—
Dependencies (require/shell.run/dofile)	—

API	Methods used
fs	—
os	—
http	—
shell	—
term	—

Code & line■by■line explanation

Line 1: `return {`
→ Returns a value to the caller “ {”.

Line 2: `updateURL_update = "gNHAVd7D",`
→ Assigns “"gNHAVd7D",” to global variable “updateURL_update”.

Line 3: `spamLimit = 5,`
→ Assigns “5,” to global variable “spamLimit”.

Line 4: `errorCodeOutdated = 279,`
→ Assigns “279,” to global variable “errorCodeOutdated”.

Line 5: `maxMessageLength = 200,`
→ Assigns “200,” to global variable “maxMessageLength”.

Line 6: `adminUser = "ragnar",`
→ Assigns “"ragnar",” to global variable “adminUser”.

Line 7: `updateURL_ui = "DWHJU4bC",`
→ Assigns “"DWHJU4bC",” to global variable “updateURL_ui”.

Line 8: `manifest = {`
→ Assigns “{” to global variable “manifest”.

Line 9: `[ "update.lua" ] = "723186416",`
→ Generic Lua instruction.

Line 10: `[ "startup.lua" ] = "2620602876",`
→ Generic Lua instruction.

Line 11: `[ "ui.lua" ] = "322429472",`
→ Generic Lua instruction.

Line 12: `},`
→ Generic Lua instruction.

Line 13: `pepper = "RAG-PEPPER-2025",`
→ Assigns “"RAG-PEPPER-2025",” to global variable “pepper”.

Line 14: `key = "RAGNAR123456789KEYULTRA2025",`
→ Assigns “"RAGNAR123456789KEYULTRA2025",” to global variable “key”.

Line 15: `expectedStartupVersion = "7.1.0",`
→ Assigns “"7.1.0",” to global variable “expectedStartupVersion”.

Line 16: `tamperAction = "error",`
→ Assigns “"error",” to global variable “tamperAction”.

Line 17: `pwdHashRounds = 512,`
→ Assigns “512,” to global variable “pwdHashRounds”.

Line 18: `autoSeal = true,`
→ Assigns “true,” to global variable “autoSeal”.

Line 19: `askUpdateAtBoot = true,`
→ Assigns “true,” to global variable “askUpdateAtBoot”.

Line 20: `errorCodeTamper = 163,`
→ Assigns “163,” to global variable “errorCodeTamper”.

Line 21: `updateURL_config = "jK7srvyY",`
→ Assigns ""jK7srvyY"," to global variable "updateURL_config".

Line 22: `updateURL_startup = "m7wpD8wF",`
→ Assigns ""m7wpD8wF"," to global variable "updateURL_startup".

Line 23: `protocol = "ragnarnet",`
→ Assigns ""ragnarnet"," to global variable "protocol".

Line 24: `adminCode = "2013.2013",`
→ Assigns ""2013.2013"," to global variable "adminCode".

Line 25: `outdatedAction = "error",`
→ Assigns ""error"," to global variable "outdatedAction".

Line 26: `spamResetTime = 300,`
→ Assigns "300," to global variable "spamResetTime".

Line 27: `}`
→ Generic Lua instruction.

Module: ui.lua

Detected role: User interface, Terminal/UI

Declared functions	fill, footBar, titleBar, trunc, ui.drawUI, ui.showMessages
Dependencies (require/shell.run/dofile)	—

API	Methods used
fs	—
os	—
http	—
shell	—
term	clear, clearLine, getSize, isColor, setBackgroundColor, setCursorPos, setTextColor, write

Code & line■by■line explanation

Line 1: `-- ui.lua : RagnarNet UI (adapté 7.2.x)`
→ Developer comment.

Line 2: `-- Exporte: ui.drawUI(username, isAdmin, w, h, version)`
→ Developer comment.

Line 3: `--  ui.showMessages(messages, uiHeight, blacklist, adminUser)`
→ Developer comment.

Line 5: `local ui = {}`
→ Declares local variable “ui” and initializes it to “{}”.

Line 7: `-- Couleurs sûres (fallback si écran non couleur)`
→ Developer comment.

Line 8: `local HAS_COLOR = term.isColor and term.isColor()`
→ Declares local variable “HAS_COLOR” and initializes it to “term.isColor and term.isColor()”.

Line 9: `local C_BG  = HAS_COLOR and colors.lightGray or colors.white`
→ Declares local variable “C_BG” and initializes it to “HAS_COLOR and colors.lightGray or colors.white”.

Line 10: `local C_TITLE  = HAS_COLOR and colors.blue  or colors.black`
→ Declares local variable “C_TITLE” and initializes it to “HAS_COLOR and colors.blue or colors.black”.

Line 11: `local C_TEXT  = HAS_COLOR and colors.white  or colors.black`
→ Declares local variable “C_TEXT” and initializes it to “HAS_COLOR and colors.white or colors.black”.

Line 12: `local C_MUTE  = HAS_COLOR and colors.gray  or colors.black`
→ Declares local variable “C_MUTE” and initializes it to “HAS_COLOR and colors.gray or colors.black”.

Line 13: `local C_WARN  = HAS_COLOR and colors.orange  or colors.black`
→ Declares local variable “C_WARN” and initializes it to “HAS_COLOR and colors.orange or colors.black”.

Line 14: `local C_BAD  = HAS_COLOR and colors.red  or colors.black`
→ Declares local variable “C_BAD” and initializes it to “HAS_COLOR and colors.red or colors.black”.

Line 16: `local function fill(x1,y1,x2,y2,c)`
→ Generic Lua instruction.

Line 17: `paintutils.drawFilledBox(x1,y1,x2,y2,c)`
→ Generic Lua instruction.

Line 18: `end`
→ End of block (function/condition/loop).

Line 20: `local function titleBar(w, title)`
→ Generic Lua instruction.

Line 21: `paintutils.drawLine(1, 1, w, 1, C_TITLE)`
→ Generic Lua instruction.

Line 22: `term.setCursorPos(2,1)`
→ Terminal display interaction (UI).

Line 23: `term.setTextColor(C_TEXT)`
→ Terminal display interaction (UI).

Line 24: `term.write(title or "")`
→ Terminal display interaction (UI).

Line 25: `end`
→ End of block (function/condition/loop).

Line 27: `local function footBar(w, h, text)`
→ Generic Lua instruction.

Line 28: `  paintutils.drawLine(1, h-2, w, h-2, C_MUTE)`
→ Generic Lua instruction.

Line 29: `  term.setCursorPos(2, h-1)`
→ Terminal display interaction (UI).

Line 30: `  term.setTextColor(C_MUTE)`
→ Terminal display interaction (UI).

Line 31: `  term.write(text or "")`
→ Terminal display interaction (UI).

Line 32: `end`
→ End of block (function/condition/loop).

Line 34: `local function trunc(s, maxw)`
→ Generic Lua instruction.

Line 35: `  if #s <= maxw then return s end`
→ Generic Lua instruction.

Line 36: `  return s:sub(1, math.max(0, maxw-1)) .. "?"`
→ Returns a value to the caller “ s:sub(1, math.max(0, maxw-1)) .. "?"”.

Line 37: `end`
→ End of block (function/condition/loop).

Line 39: `-- Public: dessine l'UI statique (cadre, barres, bouton arrêter)`
→ Developer comment.

Line 40: `function ui.drawUI(username, isAdmin, w, h, version)`
→ Declares function “ui.drawUI” with parameters “username, isAdmin, w, h, version”.

Line 41: `  term.setBackgroundColor(colors.black)`
→ Terminal display interaction (UI).

Line 42: `  term.clear()`
→ Terminal display interaction (UI).

Line 43: `  fill(1,1,w,h,C_BG)`
→ Generic Lua instruction.

Line 45: `  local head = " RagnarNet UI "`
→ Declares local variable “head” and initializes it to “" RagnarNet UI "”.

Line 46: `  if version then head = head .. "v"..tostring(version).." " end`
→ Generic Lua instruction.

Line 47: `  titleBar(w, head)`
→ Generic Lua instruction.

Line 49: `  -- bande inférieure (zone d'aide)`
→ Developer comment.

Line 50: `  local who = "Connecté en tant que " .. (username or "?") .. (isAdmin and " [ADMIN]" or "")`
→ Declares local variable “who” and initializes it to “"Connecté en tant que " .. (username or "?") .. (isAdmin and " [ADMIN]" or "")”.

Line 51: `  footBar(w, h, who)`
→ Generic Lua instruction.

Line 53: `  -- bouton arrêt (click: y==1, x>=w-12)`
→ Developer comment.

Line 54: `  term.setCursorPos(math.max(1, w-12), 1)`
→ Terminal display interaction (UI).

Line 55: `  term.setTextColor(C_BAD)`
→ Terminal display interaction (UI).

Line 56: `  term.write("[ARRETER]")`
→ Terminal display interaction (UI).

Line 57: `end`
→ End of block (function/condition/loop).

Line 59: `-- Public: affiche la liste des messages dans la zone centrale`
→ Developer comment.

Line 60: `-- messages = { {id=1, from="u", text="...", admin=false}, ... }`
→ Developer comment.

Line 61: `function ui.showMessages(messages, uiHeight, blacklist, adminUser)`
→ Declares function “ui.showMessages” with parameters “messages, uiHeight, blacklist, adminUser”.

Line 62: `  local w, h = term.getSize()`
→ Terminal display interaction (UI).

Line 63: `  -- on nettoie la zone centrale (lignes 2..h-3)`
→ Developer comment.

Line 64: `  for y = 2, (h-3) do`
→ FOR loop: iterates over “y = 2, (h-3)”.

Line 65: `    term.setCursorPos(2, y)`
→ Terminal display interaction (UI).

Line 66: `    term.clearLine()`
→ Terminal display interaction (UI).

Line 67: `  end`
→ End of block (function/condition/loop).

Line 69: `  local maxWidth = math.max(1, w - 4) -- marge à gauche/droite`
→ Declares local variable “maxWidth” and initializes it to “math.max(1, w - 4) -- marge à gauche/droite”.

Line 70: `  local start = math.max(1, #messages - uiHeight + 1)`
→ Declares local variable “start” and initializes it to “math.max(1, #messages - uiHeight + 1)”.

Line 72: `  for i = start, #messages do`
→ FOR loop: iterates over “i = start, #messages”.

Line 73: `    local m = messages[i]`
→ Declares local variable “m” and initializes it to “messages[i]”.

Line 74: `    local line = (i - start) + 2 -- commence sous la barre titre`
→ Declares local variable “line” and initializes it to “(i - start) + 2 -- commence sous la barre titre”.

Line 76: `    -- couleur par type`
→ Developer comment.

Line 77: `    if m and m.admin then`
→ IF condition: executes the block if “m and m.admin” is true.

Line 78: `      term.setTextColor(C_WARN)`
→ Terminal display interaction (UI).

Line 79: `    elseif m and m.from and blacklist and blacklist[m.from] then`
→ ELSEIF branch: alternative if “m and m.from and blacklist and blacklist[m.from]” is true.

Line 80: `      term.setTextColor(C_BAD)`
→ Terminal display interaction (UI).

Line 81: `    else`
→ ELSE branch: default case when previous conditions are false.

Line 82: `      term.setTextColor(C_TEXT)`
→ Terminal display interaction (UI).

Line 83: `    end`
→ End of block (function/condition/loop).

Line 85: `    local id   = tostring(m.id or i)`
→ Declares local variable “id” and initializes it to “tostring(m.id or i)”.

Line 86: `    local from = tostring(m.from or "?")`
→ Declares local variable “from” and initializes it to “tostring(m.from or "?")”.

Line 87: `    local txt  = tostring(m.text or "")`
→ Declares local variable “txt” and initializes it to “tostring(m.text or "")”.

Line 89: `    local raw = "["..id.."] "..from..": "..txt`
→ Declares local variable “raw” and initializes it to ""["..id.."] "..from..": "..txt”.

Line 90: `    local out = trunc(raw, maxWidth)`
→ Declares local variable “out” and initializes it to “trunc(raw, maxWidth)”.

Line 92: `    term.setCursorPos(2, line)`
→ Terminal display interaction (UI).

Line 93: `    term.write(out)`
→ Terminal display interaction (UI).

Line 94: `  end`
→ End of block (function/condition/loop).

Line 95: `end`

→ End of block (function/condition/loop).

Line 97: `return ui`

→ Returns a value to the caller “ ui”.

Database (users.db)

Unable to read DB: file is not a database

Security recommendations

• Sign or hash (SHA■256) downloaded packages (http.*) and verify prior to install.
• Restrict fs.delete and fs.open('w') to allow■listed paths.
• Log critical operations (delete, reboot, update).
• Wrap network/disk I/O with pcall/xpcall.
• Load configuration as read■only for non■privileged modules.