TECHNICAL EXPLAINER

Kali Tools GUI

כיצד המערכת עובדת — שרת MCP · שכבת הסוכנים · בסיס הנתונים

מסמך הסבר טכני | MCP + Agents + DB | אבטחת מידע

מגיש: **הראלי דודאי**

ת.ז.: 027225721

דוא"ל: hareli26@gmail.com

מוסד: **מכללת HackerU**

קורס: אדריכלות AI

מנחה: שחף

תאריך: יולי 2026

תוכן עניינים

1. סקירה כללית — כיצד החלקים מתחברים
2. שרת ה-MCP — התקנה, תפקיד והכלים שנחשפים
3. שכבת הסוכנים — מה כל סוכן עושה ואיך הוא פועל
4. בסיס הנתונים — מבנה, טבלאות וזרימת נתונים
5. זרימה מקצה לקצה (End-to-End)

1. סקירה כללית — כיצד החלקים מתחברים

Kali Tools GUI היא מערכת גרפית (Web) המאחדת את כלי ה-Kali Linux מאחורי ממשק אחד, עם שכבת סוכני AI מסוג Purple-Team. השרת כתוב ב-Python בספרייה הסטנדרטית בלבד (*אפס תלויות*) ורץ כ-root כדי שכל הכלים יעבדו. המסמך הזה מסביר לעומק שלושה רכיבים: **שרת ה-MCP**, **שכבת הסוכנים**, ו**בסיס הנתונים**.

```
┌──────────────────────────────────────────────────────────────────┐
│  דפדפן (SPA)          לקוח MCP (Claude Desktop / Claude Code)     │
│      │  HTTP                      │  stdio                        │
│      ▼                            ▼                               │
│  ┌──────────────────────────────────────────────────────────┐    │
│  │  server.py  (Python stdlib, root)  —  REST API :8777      │    │
│  │                                                           │    │
│  │   agents.py     Planner · Verifier · Reporter             │    │
│  │   bluered.py    Broker · Blue Team · Learning · Fixer     │    │
│  │   db.py         SQLite  (kaligui.db)                      │    │
│  │   obsidian.py   ייצוא כספת Markdown                       │    │
│  │        │  subprocess (argv, ללא shell)                    │    │
│  │        ▼                                                  │    │
│  │   כלי Kali אמיתיים  (nmap, nikto, sqlmap, ... 72 כלים)    │    │
│  └──────────────────────────────────────────────────────────┘    │
└──────────────────────────────────────────────────────────────────┘
```

שני "לקוחות" מדברים עם אותו שרת: הדפדפן (בני אדם) ולקוח MCP (מודל AI). שניהם משתמשים באותו REST API — ה-MCP הוא רק גשר דק שמאפשר ל-Claude להפעיל את המערכת בשיחה.

2. שרת ה-MCP — התקנה, תפקיד והכלים שנחשפים

2.1 מה זה MCP ולמה הוא כאן

MCP (Model Context Protocol) הוא פרוטוקול פתוח שמאפשר למודל שפה (כמו Claude) להפעיל "כלים" חיצוניים בשיחה. במערכת שלנו, שרת ה-MCP חושף את יכולות ה-Kali Tools GUI כ-*כלים ש-Claude יכול לקרוא להם* — כך אפשר לומר בשפה טבעית "הרץ nmap על 192.168.1.10" או "תכנן בדיקת חדירות ל-example.com", וה-AI מפעיל את המערכת בפועל.

2.2 ארכיטקטורה — לקוח דק מעל ה-REST API

שרת ה-MCP (`mcp/kali_gui_mcp.py`) בנוי עם **FastMCP** והוא **לקוח דק**: הוא *אינו* מריץ כלים בעצמו, אלא מתרגם כל קריאת-כלי לבקשת HTTP אל השרת הראשי (`http://127.0.0.1:8777`). כך כל הלוגיקה, האבטחה והביקורת נשארות במקום אחד, וה-MCP רק "מגשר".

```
Claude  ──(MCP/stdio)──►  kali_gui_mcp.py  ──(HTTP)──►  server.py :8777  ──►  כלי Kali
```

2.3 התקנה

ההתקנה אוטומטית כחלק ממתקין השרת (`deploy/install-vps.sh`, שלב `[3b]`), שיוצר סביבת Python וירטואלית ומתקין את התלות היחידה (`mcp>=1.2.0`):

```
python3 -m venv /opt/kali-gui/.venv
/opt/kali-gui/.venv/bin/pip install -r /opt/kali-gui/mcp/requirements.txt
```

לאחר מכן מחברים את השרת ללקוח MCP. ב-**Claude Code (CLI)**:

```
claude mcp add kali-tools-gui \
  --env KALIGUI_URL=http://127.0.0.1:8777 \
  -- /opt/kali-gui/.venv/bin/python /opt/kali-gui/mcp/kali_gui_mcp.py
```

ב-**Claude Desktop** — הוספה ל-`claude_desktop_config.json` תחת `mcpServers` עם ה-`command`, ה-`args` וה-`env` המקבילים.

2.4 הכלים שה-MCP חושף

שרת ה-MCP חושף 11 כלים המכסים את כל יכולות המערכת:

כלי MCP	מה הוא עושה
`list_tools(category?)`	רשימת הכלים בקטלוג — שם, בינארי, קטגוריה וסטטוס התקנה
`tool_options(tool_id)`	השדות/אפשרויות של כלי מסוים (כדי לדעת מה להעביר ל-run_tool)
`run_tool(tool_id, values)`	

	הרצת כלי בודד עם ערכים, והחזרת הפלט
`plan(intent, target)`	תכנון בדיקה מכוונה בשפה חופשית — מחזיר שלבים, בלי להריץ
`run_mission(intent, target)`	תכנון + הרצת משימה מלאה (Executor+Verifier+Reporter) → דוח Markdown
`run_purple(intent, target)`	הרצת Purple-Team מלאה → איומים, הגנות ודוח
`get_fix(signature)`	הצגת תוכנית התיקון לאיום (הפקודות + רמת הסיכון) — בלי להחיל
`apply_fix(signature, confirm)`	החלת התיקון על המארח — **מסרב ללא confirm=True** (אישור אנושי)
`export_obsidian()`	ייצוא כל הדוחות והידע לכספת Obsidian (Markdown)
`dashboard()`	מצב חי: כלים מותקנים, מודיעין נצבר, פעילות אחרונה
`knowledge()`	בסיס הידע הנצבר — חתימות ממצאים שנראו לאורך הרצות

אבטחה: הכלי `apply_fix` — שמשנה את מערכת ההפעלה — מסרב לפעול אלא אם הועבר `confirm=True` במפורש. כך גם דרך ה-AI נדרש אישור אנושי לפני כל שינוי מערכת.

2.5 גישה לשרת מרוחק (VPS)

האפליקציה על ה-VPS מאזינה ל-`127.0.0.1` בלבד (מאחורי התחברות Google). כדי לאפשר ל-MCP המקומי לגשת אליה בבטחה, פותחים מנהרת SSH ומשאירים את ה-MCP מקומי:

```
ssh -L 8777:127.0.0.1:8777 root@<VPS_IP>
# ואז KALIGUI_URL=http://127.0.0.1:8777 עובר דרך המנהרה
```

כך ה-MCP ניגש ישירות לאפליקציה דרך המנהרה המוצפנת, בלי לעקוף את שכבת ההזדהות.

2.6 דוגמת שימוש (בתוך Claude)

- "השתמש ב-kali-tools-gui: הרץ nmap על 192.168.1.10"
- "תכנן בדיקת חדירות ל-example.com והצג לי את השלבים"
- "הרץ Purple-Team על scanme.nmap.org ותן לי את הדוח"

3. שכבת הסוכנים — מה כל סוכן עושה ואיך הוא פועל

3.1 עיקרון: מנוע מבוסס-חוקים (Playbooks), אפס תלויות

הסוכנים אינם "קופסה שחורה" — הם **מבוססי-חוקים** (`agents.py`): אוסף **Playbooks** שקל לביקורת ולעריכה. כל Playbook מכיל מילות-מפתח (עברית+אנגלית) ופונקציית `build` שבונה רשימת שלבים. יש 14 playbooks (DNS, תת-דומיינים, סריקת פורטים, בדיקת חדירות, SMB/Windows, Active Directory, API, מיפוי רשת, SSL/TLS, SQL Injection, WordPress, OSINT, חיפוש אקספלויטים, פורנזיקה/סטגנוגרפיה) ו-Playbook כללי כברירת מחדל. *אופציונלית*, אם מוגדר מודל Ollama מקומי, הדוח משתדרג לניסוח מבוסס-LLM — אך המערכת עובדת מצוין גם בלעדיו.

3.2 ארבעת סוכני הליבה וזרימת המשימה

```
כוונה + מטרה
   |
   ▼
[ Planner ]  רשימת שלבים ← Playbook כוונה בשפה חופשית ← התאמת
   |         (כל שלב: כלי + פקודה מוכנה + "למה" + הצעת פירוש)
   ▼   ← אישור המשתמש
[ Executor ] מריץ כל שלב ברצף כ-Job (subprocess, argv ללא shell, timeout לכל שלב)
   |
   ▼
[ Verifier ] מנתח פלט + קוד יציאה → verdict + חילוץ ממצאים (regex)
   |         verdict ∈ { תקין · ממצאים · אזהרה · נכשל }
   ▼
[ Reporter ] מרכיב דוח Markdown: תקציר, ממצאים עיקריים, פירוט שלבים
```

סוכן	קלט	פעולה	פלט
Planner agents.py	כוונה + מטרה	מתאים Playbook לפי מילות מפתח ובונה שלבים ממוקדים	תוכנית שלבים (כלי + פקודה + נימוק)
Executor server.py	שלבי התוכנית	מריץ כל שלב כ-subprocess — Job כמערך `argv` ללא shell (הגנה מהזרקה), עם מגבלת זמן	פלט חי + קוד יציאה לכל שלב
Verifier agents.py	פלט + קוד יציאה	מנתח את התוצאה, נותן verdict, ומחלץ ממצאים באמצעות ביטויים רגולריים	verdict + רשימת ממצאים
Reporter agents.py	כל השלבים המאומתים	מסכם לכדי דוח קריא; אופציונלית משדרג עם LLM מקומי	דוח Markdown מלא

3.3 שכבת ה-Purple Team

מעל סוכני הליבה יושבת שכבת Purple-Team (`bluered.py`) המחברת בין תקיפה להגנה:

- 🔴 **צוות אדום** — מנוע המשימה (למעלה) מריץ את הכלים ההתקפיים ומפיק ממצאים.
- 🟢 **מתווך (Broker)** — לוקח כל ממצא אדום וממפה אותו לכלל הגנה מתוך `DEFENSE_KB`, מאחד כפילויות ומדרג לפי חומרה.

- 🔵 **צוות כחול** — לכל איום מפיק: פעולות הגנה, המלצת זיהוי/ניטור, שיוך **MITRE ATT&CK** ותצורה לדוגמה.
- 🧠 **למידה מתמשכת** — כל הרצה מעדכנת את בסיס הידע (`signatures`): כמה פעמים נראה כל ממצא, מתי לראשונה ומתי לאחרונה.
- 🟣 **מתזמר (Orchestrator)** — מריץ את הזרימה כולה: אדום → Broker → כחול → למידה → דוח Purple.

3.4 סוכן מתקן (Remediation / Fixer)

הסוכן המתקן מיישם בפועל את הגנות הצוות הכחול (למשל התקנת `fail2ban`, עדכוני אבטחה), אך תמיד תחת בקרות בטיחות: **אישור אנושי חובה**, גיבוי אוטומטי של קונפיגים לפני שינוי, ורישום ביומן הביקורת. תיקונים מסוכנים (כיבוי סיסמת SSH, שינויי firewall) מוגדרים כ**הנחיה בלבד** ואינם רצים אוטומטית — כדי למנוע נעילה עצמית של השרת.

> **עיקרון מנחה:** הסוכנים בונים פקודות אמיתיות ומריצים כלים אמיתיים, אך כל פעולה התקפית עוברת שלב אישור, וכל שינוי מערכת (תיקון) דורש אישור מפורש ונרשם בביקורת.

3.9 🍯 סוכן החיישן — ואיך הסוכנים לומדים מתקיפות אמיתיות

עד כה הסוכנים למדו רק מתקיפות *שאנחנו* יזמנו — הצוות האדום סרק, הצוות הכחול הגיב. זה מעגל סגור: המערכת מלמדת את עצמה על סמך צ'קליסט קבוע שכתבנו מראש. **סוכן החיישן** (`sensor.py`) פותח אותו לעולם האמיתי.

```
🌍 תוקף אמיתי
      │
      ▼
🍯 מלכודת (שרת מוקרב נפרד) — קולטת בקשה גולמית, לא מסווגת, לא מריצה כלום
      │  ▲ הפרודקשן מושך (המלכודת לא מחזיקה מפתח לפרודקשן)
      ▼  │
🛰️ מסווג מול — סוכן החיישן ATTACK_KB (19 טכניקות)
      │
      ├──► hp_signatures      "מה תוקפים מנסים"    (התנהגות)
      │          │
      │          ├──► 🔥 hp_correlate()  ◄── signatures  "מה חלש אצלנו" (תנוחה)
      │          │
      ▼          ▼
   [Planner] מוסיף בדיקות למה שבאמת מנסים   [Blue] "מסמן "תחת תקיפה פעילה"
```

הלולאה סגורה בשלוש נקודות:

איפה	מה משתנה בפועל
`agents.plan()`	טכניקה שנצפתה ≥3 פעמים מוסיפה שלבי בדיקה. **נמדד:** תוכנית של 5 שלבים גדלה ל-8 — נוספו `sqlmap` (SQLi ×47), `gobuster` (ציד סודות ×18), `wpscan` (CMS ×9).
`agents.llm_plan()`	המודיעין החי מוזרק לפרומפט, כך שה-LLM בוחר כלים לפי איומים אמיתיים ולא בוואקום.
`bluered.purple_report()`	סעיף "🔥 **תעדוף מיידי**" — חולשה שהמלכודת ראתה מנוצלת מסומנת כלא-תיאורטית.

ולמה זה לא מסוכן: נתוני מלכודת הם *קלט של תוקף*. לכן ההשפעה מוגבלת ל**מה בודקים**, אף פעם לא למה שעושים לעצמנו: רק כלי גילוי (`hydra` וכלים תוקפניים מוחרגים במכוון — אחרת תוקף יציף brute-force ויגרום לנו לנעול את חשבונותינו), נדרשות 3 תצפיות (שבקשה מזויפת אחת לא תטה תכנון), עד 3 שלבים נוספים, כל שלב מסומן במקורו, המשתמש מאשר — ו**אף פעם אין תיקון אוטומטי מנתוני מלכודת**.

4. בסיס הנתונים — מבנה, טבלאות וזרימת נתונים

4.1 טכנולוגיה ועקרונות

בסיס הנתונים (`db.py`) הוא **SQLite** יחיד (`kaligui.db`) המשתמש במודול `sqlite3` של הספרייה הסטנדרטית — *אפס תלויות חיצוניות*. הוא מחליף את קבצי ה-JSON השטוחים הישנים, ובהפעלה הראשונה

מייבא אוטומטית נתונים קיימים (/knowledge.json / activity.json / audit.log / reports) כדי ששום מידע לא יאבד. הגישה מוגנת בנעילה (`RLock`) לבטיחות ריבוי-תהליכונים.

4.2 הטבלאות

טבלה	תפקיד	עמודות עיקריות
signatures	בסיס הידע הנלמד — שורה לכל חתימת ממצא	sig (PK), name, severity, count, first_seen, last_seen
meta	מפתח/ערך גלובלי (למשל מונה ההרצות)	k (PK), v
activity	יומן פעילות — היסטוריית משימות ו-Purple	id, ts, whenstr, type, intent, target, user, status, data(JSON)
audit	יומן ביקורת אבטחה — מי עשה מה	id (PK), ts, user, action, detail
reports	דוחות Markdown שמורים	id (PK), kind, intent, target, ts, whenstr, report
playbooks	חוקי סוכן מותאמים (data-driven, מעורך ה-UI)	id (PK), data (JSON)
roles	הרשאות תפקידים (RBAC) — תפקיד לכל משתמש	email (PK), role (admin/operator/viewer)

4.3 מי כותב ומי קורא

פעולה במערכת	פונקציה ב-db.py	טבלה
סיום Purple-Team → למידה	update_kb()	signatures + meta
כל הרצה נרשמת ליומן	add_activity()	activity
כל פעולה רגישה (הרצה/תיקון/משתמש)	add_audit()	audit
שמירת דוח בסיום משימה	save_report()	reports
דשבורד / מסך ידע	load_kb() · stats() · list_activity()	קריאה
ייצוא Obsidian	all_reports_full()	קריאה

הטבלאות `activity` נשמרת עד 500 רשומות אחרונות (גיזום אוטומטי), וה- `audit` נצברת במלואה לצורכי ביקורת.

4.4 גישה, גיבוי ותחזוקה

הקובץ נמצא ב- `/opt/kali-gui/kaligui.db` . אפשר לבדוק אותו ישירות:

```
sqlite3 /opt/kali-gui/kaligui.db ".tables"
sqlite3 /opt/kali-gui/kaligui.db "SELECT name,severity,count FROM signatures ORDER
BY count DESC;"
```

גיבוי הוא פשוט העתקת הקובץ (למשל בתוך ה-tar של נתוני האפליקציה). נקודת קצה `GET /api/dbstats` מחזירה ספירת רשומות לכל טבלה לבדיקה מהירה.

5. זרימה מקצה לקצה (End-to-End)

דוגמה: המשתמש (או Claude דרך MCP) מבקש "בדיקת חדירות ל-example.com".

```
1. בקשה  →  Planner מזהה Playbook "בדיקת חדירות" ובונה שלבים
2. אישור →  Executor מריץ כל שלב (nmap → whatweb → nikto → nuclei ...)
3. לכל שלב: Verifier נותן verdict + מחלץ ממצאים
4. Broker הגנות → ממפה ממצאים (DEFENSE_KB + MITRE),  Blue Team מפרט
5. update_kb() מעדכן signatures,  add_activity() רושם ליומן
6. Reporter מפיק דוח,  save_report() שומר ב-DB
7. obsidian.export() מסנכרן לכספת,  audit() רושם מי הריץ
8. הדוח מוצג בדפדפן / מוחזר ל-Claude דרך ה-MCP
```

כל שמונת השלבים משתמשים באותו REST API — בין אם ההפעלה הגיעה מהדפדפן ובין אם מ-Claude דרך ה-MCP. כך המערכת אחידה, ניתנת לביקורת, ובעלת מקור אמת אחד.

> בקצרה: **ה-MCP** מאפשר ל-AI להפעיל את המערכת בשיחה; **הסוכנים** הופכים כוונה בשפה חופשית לבדיקה אמיתית, מאמתים ומדווחים, וממפים כל ממצא להגנה; **בסיס הנתונים** שומר את הידע הנלמד, ההיסטוריה, הדוחות והביקורת — הכל ב-SQLite אחד ללא תלויות.