SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

Tesi di Laurea Triennale in
TECNOLOGIE INFORMATICHE

Disegno e Implementazione di un sistema di aggiornamento per Open CAPWAP

Relatore:
Prof. Massimo Bernaschi

Candidato:
Donato Capitella

Anno Accademico 2009/2010

Indice

Introduzione

Internet (con la “I” maiuscola, per distinguerla dalla sua tecnologia) nasce nel 1982 come un'estensione della preesistente rete militare ARPANET dedicata alle applicazioni civili e al mondo accademico. Nel giro di pochi decenni, l'Internet degli anni ottanta si trasformerà nella moderna biblioteca di Alessandria, una fonte inesauribile di conoscenza condivisa a cui tutti possono partecipare e che ciascuno può contribuire ad accrescere. Ma la Rete, lungi dall'essere solo una sconfinata fonte di conoscenza, è oggi diventata la piazza in cui vengono offerti i più disparati servizi, dai siti delle banche fino alle reti sociali in cui interagiscono telematicamente gruppi di persone connesse tra loro da diversi legami sociali, che vanno dai rapporti di lavoro alla condivisione di interessi comuni.

Ed è in questo scenario che gli utenti di Internet avvertono più che mai la necessità di potersi collegare ovunque, muovendosi liberamente senza essere più legati al concetto di “cavo”. Per questo motivo le reti senza fili o wireless, nate con pretese molto ridotte e adatte al più a servire case e uffici, oggi si stanno rapidamente estendendo fino a coprire intere città. Un esempio è il progetto *Provincia Wi-Fi*: esso mira ad offrire connettività wireless gratuita ad Internet nei principali luoghi pubblici della Provincia di Roma, quali biblioteche, piazze e parchi.

La nascita di reti wireless di grandi dimensioni pone un serio problema: in che modo dispiegare e gestire il crescente numero degli apparati necessario ad offrire l'accesso alla rete? Lo standard utilizzato attualmente per le reti wireless, l'IEEE 802.11, manca di specifiche per affrontare tali problematiche. Per questa ragione, nel corso degli anni, numerosi produttori hanno sviluppato soluzioni proprietarie e incompatibili tra di loro. Per cercare di mettere ordine in questo coacervo di tecnologie incompatibili, l'*Internet Engeneering Task Force* (**IETF**) ha proposto il protocollo *Control and Provisioning of Wireless Access Points* (**CAPWAP**). Tale protocollo consente di gestire, tramite un computer centrale (*Access Controller*, nel seguito AC), un numero anche elevato di apparati wireless (*Wireless Termination Point*, nel seguito WTP).

Nel mio tirocinio ho lavorato su OpenCAPWAP, un'implementazione libera del protocollo CAPWAP attualmente utilizzata per la gestione degli Access Point del progetto Provincia Wi-Fi. Il tirocinio si è articolato in tre fasi: la prima relativa allo studio della tecnologia e al debugging della release 0.91 di Open CAPWAP; la seconda relativa al disegno e all'implementazione di un sistema di aggiornamento del software di controllo degli Access Point; la terza relativa al

test e al collaudo della versione 0.93.2, che incorpora le modifiche da me apportate.

Questa relazione di tirocinio è strutturata nei seguenti capitoli:

- *Capitolo 1, Reti Wireless e Protocollo CAPWAP*: breve analisi dello standard IEEE 802, delle problematiche relative alla reti wireless di grandi dimensioni e delle soluzioni offerte dal protocollo CAPWAP.
- *Capitolo 2, Struttura di OpenCAPWAP*: introduzione all'architettura software di OpenCAPWAP. In particolare si analizzano i thread dell'AC e del WTP e l'interfaccia dell'AC verso le applicazioni esterne, esponendo in dettaglio la struttura dei messaggi.
- *Capitolo 3, Debugging di OpenCAPWAP*: analisi degli strumenti usati per il debugging e dei risultati ottenuti. Viene qui analizzata l'architettura dei due strumenti scelti per il debugging: gdb e Valgrind. Infine, vengono presentate delle statistiche sui bug individuati e risolti. Questa fase del tirocinio ha portato al rilascio su *Sourceforge.org* della release 0.92 di OpenCAPWAP.
- *Capitolo 4, Sistema di aggiornamento dei WTP*: fase principale del tirocinio relativa al disegno e implementazione del sistema di aggiornamento dei WTP. Vengono presentati i due moduli software aggiunti a OpenCAPWAP - *WTP Update Agent* (**WUA**) e *WTP Update Manager* (**WUM**) – i nuovi messaggi aggiunti al protocollo e la struttura dei pacchetti di aggiornamento **CUP** (*CAPWAP Update Package*). Questa fase ha portato allo sviluppo della versione 0.93.2 di OpenCAPWAP.
- *Capitolo 5, Test e conclusioni*: in questo capitolo vengono descritti i test effettuati sul progetto, i risultati raggiunti e i possibili scenari di applicazioni e sviluppi futuri.
- *Appendice, Elenco dettagliato dei bug*: in questa appendice ho riportato una descrizione completa di tutti i bug da me riscontrati e corretti.

Capitolo 1
Reti Wireless e Protocollo CAPWAP

"Rem tene, verba sequentur."
*-- **Marco Tullio Cicerone***

La prima parte di questo capitolo è dedicata all'analisi di una delle principali tecnologie che sta guidando la trasformazione delle reti di computer via cavo in reti "senza fili": lo standard 802.11. La seconda parte affronta, invece, le problematiche correlate alle reti wireless di grandi dimensioni e il modo in cui esse possono essere affrontate con il protocollo CAPWAP.

1.1 WLAN e Standard 802.11

La sigla 802.11 denota una serie di standard appartenenti alla famiglia IEEE 802: tale famiglia definisce le specifiche per le reti locali (LAN). In particolare, l'IEEE 802 si concentra sui due livelli più bassi del modello ISO/OSI, definendo sia lo strato fisico (PHY) che quello MAC (*Medium Access Control*) dei diversi sotto-standard.

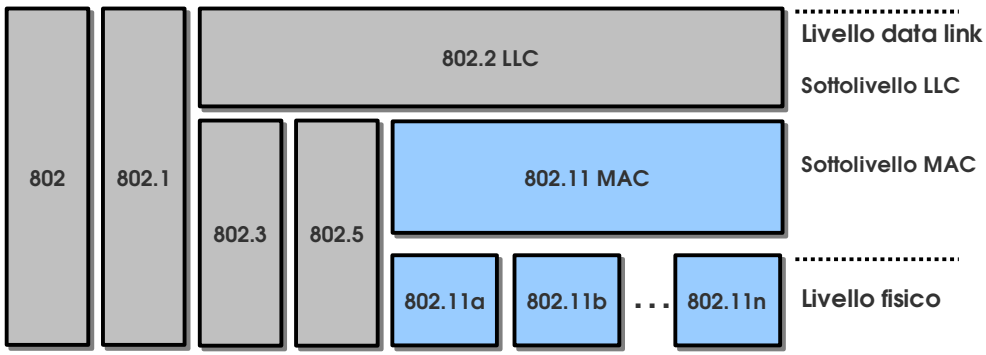

Figura 1: Collocamento dell'802.11 all'interno della famiglia IEEE 802

Dal 1997, all'interno di questa famiglia, l'IEEE ha iniziato a definire gli standard per le reti locali senza fili o WLAN (Wireless LAN), raggruppati sotto la denominazione 802.11. La Figura 1 mostra come l'802.11 si inserisce all'interno dell'802.

Come accennato in precedenza, ciascuno degli standard definisce gli strati inferiori del modello ISO/OSI per una comunicazione senza fili basata su onde elettromagnetiche:

- **sottolivello MAC**: definisce i meccanismi di accesso al canale condiviso;
- **livello fisico (o PHY)**: propone diverse tecnologie fisiche di codifica e trasmissione dell'informazione (DSSS, FHSS, Infrarossi, *etc...*).

Come è evidente dalla Figura 1, tutti gli standard dell'802.11 condividono il livello MAC; si differenziano, invece, per il livello fisico e quindi per le tecnologie di trasmissione utilizzate (le quali hanno impatto sulle prestazioni).

1.1.1 Architetture di una WLAN 802.11

Una WLAN 802.11 può essere organizzata secondo due architetture differenti: **Managed** o **Ad-Hoc** [1]. Essenzialmente, la prima si basa su una stazione centrale a cui le altre stazioni fanno riferimento per comunicare tra loro ed eventualmente con una o più reti esterne (ad esempio Internet); la seconda è invece una rete senza controllo centrale.

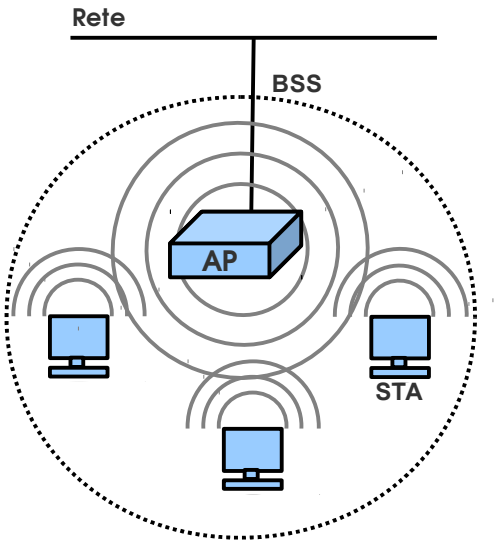

Figura 2: Architettura Managed

Il blocco principale (Figura 2) dell'architettura **Managed** è la cella, chiamata **BSS** (*Basic Service Set*): solitamente una cella contiene una o più stazioni (chiamate STA) senza fili connesse ad una stazione base, **BS** (*Base Station*) - chiamata **AP** (*Access Point*) nella terminologia 802.11. Ogni BSS ha un identificativo di 6 byte, **BSSID**, che spesso corrisponde all'indirizzo MAC dell'AP.

E' possibile collegare più AP tra loro tramite un **DS** (*Distribution System*); in questo caso si ottiene un insieme di servizi estesi o **ESS** (Extended Service Set) che unisce le STA dei diversi BSS gestiti dagli AP. Ciascun ESS è identificato da un **ESSID** di 32 caratteri ASCII. Una stazione può spostarsi liberamente all'interno di un ESS ed è in grado di cambiare il punto di accesso in modo trasparente in base alla qualità del segnale (*roaming*).

Nell'architettura **Ad-hoc** (Figura 3), la rete si forma "spontaneamente" quando due o più stazioni desiderano comunicare tra loro ma non è disponibile un'infrastruttura di rete preesistente (ad esempio, un AP). Spesso, una rete di questo tipo è indicata con la sigla **IBSS** (*Indipendent Basic Service Set*). Un'architettura di questo tipo è pensata, soprattutto, per consentire a due o più stazioni di usufruire dei servizi che loro stesse mettono a disposizione, come ad esempio la condivisione di file.

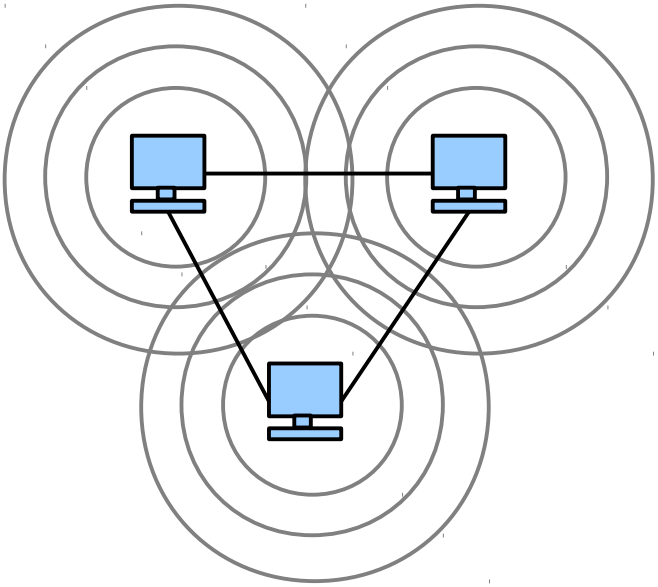

Figura 3: Architettura Ad-Hoc

1.1.2 Sottolivello MAC

Non è scopo di questo testo trattare approfonditamente il sottolivello MAC dell'802.11; si offrirà quindi solo una breve introduzione alle problematiche che esso deve affrontare e alle soluzioni adottate.

Esattamente come accade per una rete 802.3, le stazioni di una WLAN 802.11 devono coordinare il loro accesso ad un mezzo fisico condiviso: una frequenza radio. Se ciò non avvenisse (o avvenisse in modo errato), le comunicazioni delle varie stazioni si sovrapporrebbero generando un segnale corrotto e quindi inutilizzabile (**collisione**).

Il compito del sottolivello MAC è di definire le regole di accesso al mezzo fisico che una stazione deve seguire per evitare collisioni. La soluzione proposta per l'802.11 è chiamata **CSMA/CA** (*Carrier-Sense Multiple Access Protocol with Collision Avoidance*). Senza entrare nei dettagli, il protocollo si basa su due parti: rilevazione della portante e prevenzione delle collisioni. La prima parte prevede che una stazione, prima di trasmettere, sondi il mezzo fisico per rilevare se è già in corso un'altra trasmissione; in caso affermativo, dovrà astenersi dalla trasmissione. La seconda parte specifica che il protocollo mette in atto meccanismi per evitare che avvengano collisioni piuttosto che per rilevare quando ne avviene una (come, invece, si fa nell'802.3).

1.1.3 Standard attuali

Come detto, l'802.11 specifica diversi standard; essi si differenziano principalmente per il livello fisico, mentre non presentano differenze sostanziali per quanto riguarda il sottolivello MAC.

Versione	Data	Frequenza	Bitrate	Modulazione
-	Giu 1997	2.4 GHz	2 Mbit/s	DSSS
a	Sett 1999	5 GHz	54 Mbit/s	OFDM
b	Sett 1999	2.4 GHz	11 Mbit/s	DSSS
g	Giu 2003	2.4 GHz	54 Mbit/s	OFDM
y	Nov 2008	3.7 GHz	54 Mbit/s	OFDM
n	Sett 2009	2.4/5 GHz	600 Mbit/s	OFDM

Figura 4: Standard 802.11

La Figura 4 riassume le caratteristiche degli standard approvati fino al momento della scrittura di questo testo.

In particolare, lo standard 802.11n, recentemente approvato, costituisce un salto di qualità per quanto riguarda le prestazioni delle WLAN: esso, infatti, consente di ottenere un *bitrate* pari a 600 Mbit/s, un ordine di grandezza in più rispetto alle capacità dei precedenti standard.

1.1.4 Problemi delle WLAN di grandi dimensioni

Il crescente bisogno di accesso alla rete, da un lato, e l'avanzamento tecnologico degli standard wireless dall'altro, hanno determinato la nascita di WLAN di grandi dimensioni in grado di coprire aree geografiche sempre più vaste. Più vasta è l'area da servire, maggiore sarà il numero di AP da dispiegare sul territorio.

Questo rilevante dispiegamento di AP pone problemi seri per quanto riguarda la gestione e il controllo degli apparati. Ogni AP è, infatti, un dispositivo con un indirizzo IP e, come tale, deve essere opportunamente configurato e amministrato. Inoltre, parte della configurazione deve essere condivisa dai diversi AP, che potranno così operare in maniera coerente.

Per far fronte a questi problemi, i vari fornitori, nel corso degli anni, hanno sviluppato soluzioni proprietarie. Tali soluzioni, seppur basate su idee comuni, sono tra loro incompatibili.

In generale, sono state proposte due architetture: una WLAN autonoma e una centralizzata. Nella prima, ogni AP agisce in maniera indipendente dagli altri e accentra tutti i servizi relativi al proprio BSS. Nella seconda, distinguiamo tra due tipi di apparati: WTP e AC. I **WTP** (*Wireless Termination Point*) gestiscono il traffico delle varie STA associate e, al minimo, implementano lo strato fisico dell'802.11. L'**AC** (*Access Controller*) offre ai WTP servizi di controllo e configurazione (Figura 5).

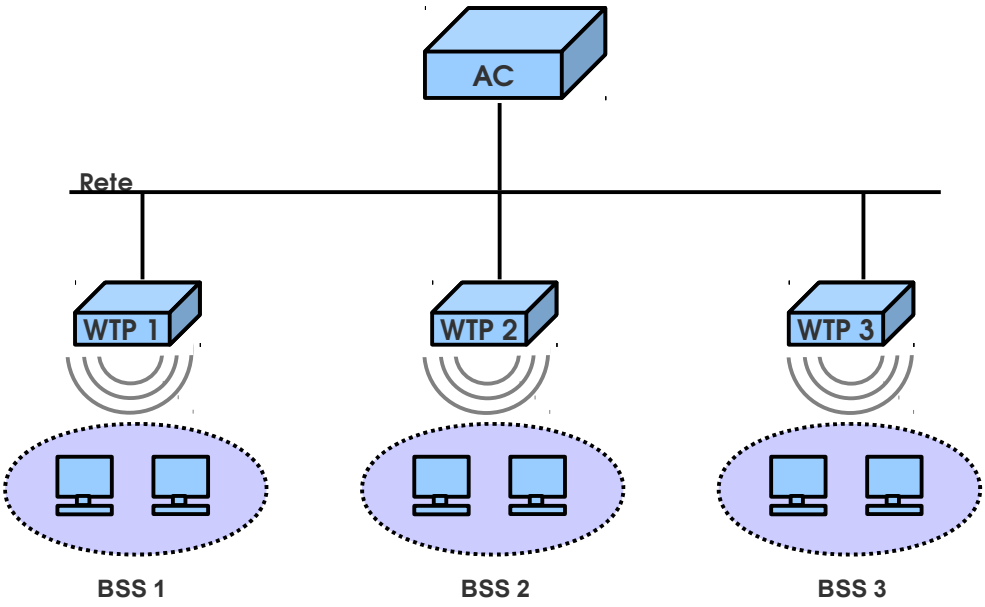

Figura 5: WLAN Centralizzata

L'architettura centralizzata è quella usata dalla maggior parte dei sistemi proprietari [2]. Tutte le soluzioni proposte si basano su due elementi comuni:

- separare le funzioni offerte dagli AP;
- aggiungere funzioni centralizzate per il controllo e il monitoraggio della rete.

La differenza tra le varie architetture centralizzate sta nella scelta di quali funzioni centralizzare e quali delegare ai singoli AP. In generale, maggiore è il numero di funzioni centralizzate, più semplici ed economici saranno gli AP da impiegare. Questo porta a infrastrutture di rete estremamente flessibili. Ad esempio, funzioni critiche dal punto di vista del tempo, come la generazione dei *beacon* e la gestione degli *acknowledgment*, possono essere delegate ai singoli AP; mentre funzioni non critiche come la selezione del canale, l'autenticazione e la cifratura possono essere centralizzate.

1.2 Protocollo CAPWAP

CAPWAP [3] (*Control And Provisioning of Wireless Access Points*) costituisce una recente iniziativa dell'IETF che ha come obiettivo quello di definire un protocollo interoperabile per la gestione centralizzata di Access Point wireless. L'architettura adottata è quindi quella centralizzata, analizzata nel paragrafo 1.1.4, i cui due principali apparati sono:

- l'**AC** (*Access Controller*): apparato dedicato alla gestione centralizzata dei vari WTP;
- il **WTP** (*Wireless Termination Point*): apparato che gestisce (al minimo) il traffico PHY 802.11 con le varie stazioni associate.

Sebbene motivato dai problemi precedentemente ricordati delle reti 802.11, il protocollo CAPWAP mira ad essere indipendente dalla specifica tecnologia radio dei WTP. Di seguito elenco i principali obiettivi che hanno guidato lo sviluppo del protocollo [4]:

- rendere il protocollo indipendente dalla tecnologia wireless;
- centralizzare l'autenticazione, la configurazione e l'implementazione delle politiche per una rete wireless;
- suddividere il traffico per poter distinguere tra traffico dati e traffico generato da operazioni di amministrazione;
- spostare il più possibile le funzioni di elaborazione sull'AC, delegando ai WTP solo le funzioni critiche dal punto di vista del tempo;
- fornire un meccanismo generale di incapsulamento e di trasporto;
- prevedere un meccanismo di reciproca autenticazione tra WTP e AC che soddisfi, oltre al requisito di autenticazione, anche quelli di confidenzialità e integrità.

Nei prossimi paragrafi introdurrò le architetture supportate da CAPWAP, il formato dei messaggi e gli stati in cui possono transitare gli apparati del sistema (cioè AC e WTP).

1.2.1 *Architetture operazionali supportate da CAPWAP*

CAPWAP, nelle specifiche di *binding* con l'IEEE 802.11 [5], prevede due tipi di architetture operazionali: **Split MAC** (SM) **e Local MAC** (LM). Esse si differenziano in base all'attribuzione delle responsabilità di alcuni servizi ai WTP piuttosto che all'AC. In entrambe, il livello fisico 802.11 è sempre gestito dai WTP.

Nella variante **Split MAC**, i WTP gestiscono solo le funzioni *real-time* del livello MAC. Funzioni *real-time* sono considerate, ad esempio, la sincronizzazione e la ritrasmissione dei frame. Tutte le altre funzioni di livello MAC non *real-time* sono invece demandate all'AC. Esempi di funzioni non *real-time* sono: l'autenticazione, la cifratura dei frame e la gestione della QoS (*Quality of Service*). Con questa architettura diminuiscono le responsabilità dei WTP e si riducono, quindi, i costi dei singoli AP.

Nella variante **Local MAC**, i WTP si assumono la responsabilità di gestire interamente il livello MAC 802.11, liberando così l'AC da questo onere. All'AC resta quindi solo il compito di gestire la configurazione dei WTP e assicurarsi che siano sempre in uno stato consistente.

1.2.2 *Pacchetti*

CAPWAP distingue i pacchetti in due tipologie: pacchetti di controllo e pacchetti di dati. Per entrambe le tipologie è previsto un formato in chiaro e uno cifrato ed autenticato. In quest'ultimo caso, si ricorre al protocollo **DTLS** (*Datagram Transport Layer Security*) che è l'equivalente del **TLS** (*Transport Layer Security*) per l'UDP.

I **pacchetti di controllo** (Figura 6) sono usati unicamente per la gestione dell'infrastruttura di rete. Tra le operazioni tipiche vi sono la raccolta delle statistiche, la gestione della configurazione e degli aggiornamenti e il mantenimento della sessione DTLS.

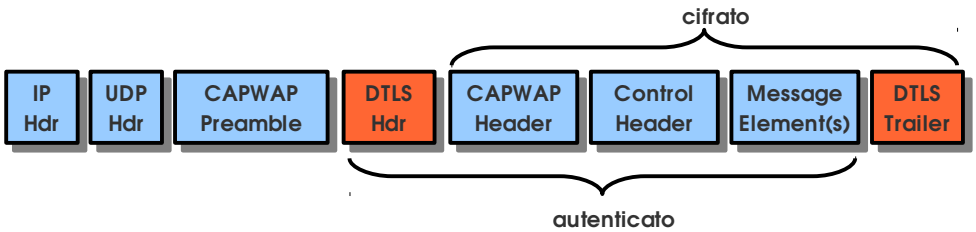

Figura 6: CAPWAP: Formato dei pacchetti di controllo

I pacchetti dati (Figura 7) servono, invece, a trasportare dati generici. Come mostrato nella figura, il traffico dello standard wireless usato viene incapsulato all'interno del pacchetto dati CAPWAP.

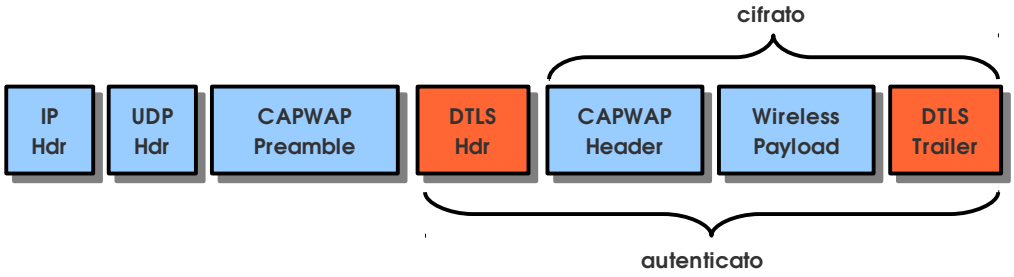

Figura 7: CAPWAP: Formato dei pacchetti dati

1.2.3 Stati del protocollo

In Figura 8 è riportato l'automa a stati finiti che descrive le transizioni di stato degli apparati durante una sessione di lavoro. Dato che il protocollo CAPWAP fa uso del DTLS, l'automa incorpora anche gli stati propri di questo protocollo. Si noti, inoltre, che l'automa di Figura 8 descrive sia gli stati degli AC che dei WTP. Non tutti gli stati, comunque, sono implementati su entrambe le apparecchiature.

Di seguito è offerta una breve descrizione degli stati e dei messaggi che determinano le transizioni:

- **Discovery**: il WTP cerca di contattare un AC inviando un messaggio di *Discovery Request* in broadcast, multicast o unicast a seconda dell'implementazione. L'AC risponde con una *Descovery Response* che notifica al WTP l'effettiva disponibilità dell'AC e l'insieme dei servizi offerti. In questa fase tutto il traffico è in chiaro.
- **Sessione DTLS**: come accennato, la macchina a stati finiti di CAPWAP incorpora la macchina a stati finiti del protocollo DTLS (*DTLS Setup*, *DTLS Connect*, *DTLS Teardown*). In questi stati avviene lo scambio delle chiavi, il *setup* della connessione cifrata ed, eventualmente, la chiusura della sessione. CAPWAP supporta l'autenticazione tramite certificati o chiavi pre-condivise, sebbene quest'ultima possibilità sia sconsigliata perché più soggetta ad attacchi.
- **Join**: in questo stato, il WTP chiede ad un determinato AC di poter usufruire dei suoi servizi inviando una *Join Request*. L'AC risponde inviando una *Join Response*; se non arriva alcuna risposta, la connessione DTLS viene abbattuta.
- **Configure**: in questo stato l'AC ha la possibilità di analizzare la configurazione del WTP, inviata tramite una *Configuration Status Request*. Nella risposta, *Configuration Status Response*, l'AC può comunicare al WTP eventuali modifiche da apportare alla sua configurazione.

- **Run**: si tratta del normale stato operativo, in cui WTP e AC rimangono a tempo indeterminato fino al verificarsi di condizioni particolari (es.: caduta della connessione). Per mantenere attiva la connessione, il WTP genera *Echo Request* ad intervalli regolari, a cui l'AC risponde con *Echo Response*. In questo stato è possibile aggiornare e/o modificare la configurazione dei WTP tramite la coppia di messaggi *Configuration Update Request* e *Configuration Update Response*.
- **Idle**: stato di inattività in cui il WTP transita, ad esempio, in seguito al fallimento di una procedura Discovery oppure in seguito all'impossibilità di attivare una sessione DTLS. Il WTP permane in questo stato per un tempo prefissato.
- **Sulking**: se la procedura di Discovery fallisce un dato numero di volte, il WTP transita nello stato Sulking. In questo stato il WTP deve tassativamente ignorare ogni messaggio che gli perviene.
- **Reset**: in caso di problemi imprevisti, le apparecchiature transitano nello stato Reset.
- **Data Check**: in questo stato il WTP comunica all'AC l'esito dello stato *Configure*.
- **Image Data**: l'AC, tramite una *Image Data Request*, può comunicare ad un WTP la necessità di eseguire un aggiornamento del *firmware*; il WTP transita quindi nello stato *Image Data* nel quale avviene la procedura di aggiornamento.

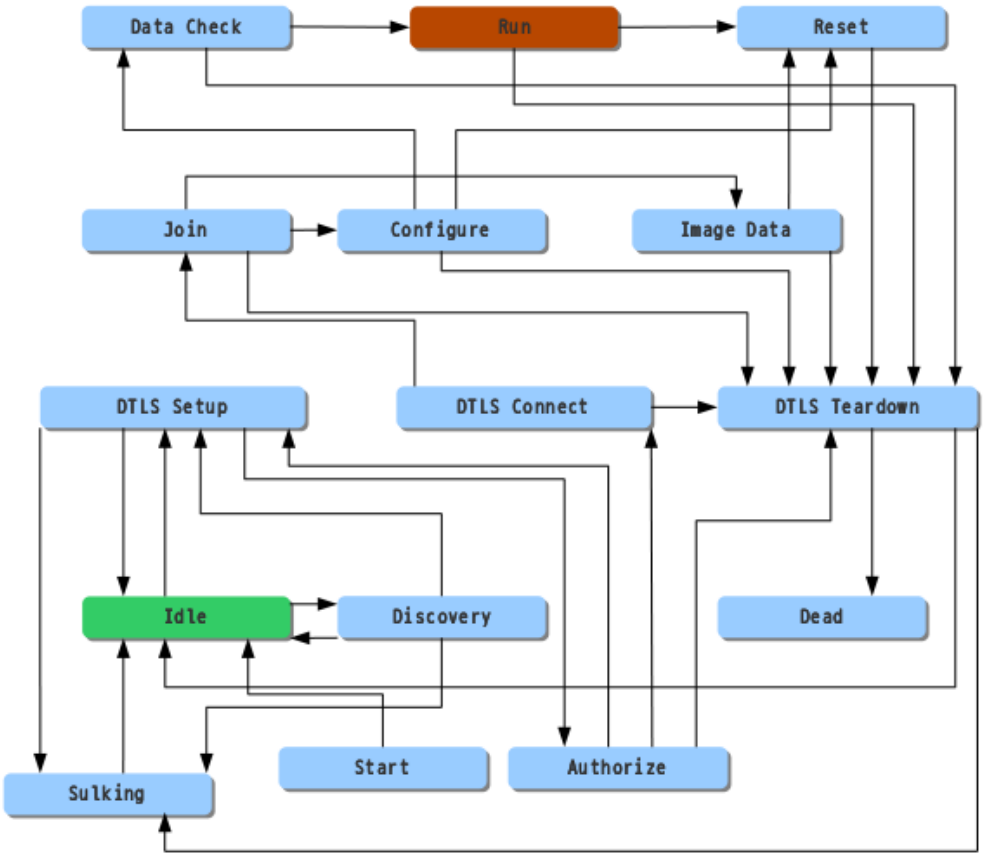

Figura 8: Macchina a Stati Finiti di CAPWAP

Capitolo 2
Struttura di OpenCAPWAP

"A man without a foundation is like a ship without a rudder and a compass, buffeted here and there by every wind."
*-- **Samuel Smiles***

In questo capitolo viene descritta la struttura di OpenCAPWAP[1]. Si tratta di un'implementazione *open-source* del protocollo CAPWAP che viene sviluppata e gestita da studenti di Informatica dell'Università "La Sapienza" in collaborazione con l'IAC (CNR), l'Università Campus Biomedico di Roma e con il CASPUR.

2.1 OpenCAPWAP

OpenCAPWAP è stato sviluppato prevalentemente per l'esecuzione su piattaforme aperte. Per la fase di sperimentazione e test è stata quindi adottata una distribuzione di Linux minimale, OpenWrt[2], sviluppata appositamente per l'utilizzo su dispositivi di tipo Access Point Wireless.

Le caratteristiche principali di OpenCAPWAP sono che:

- è sviluppato interamente in C (~ 20.000 LoC – *Lines of Code*);
- offre due demoni da usare rispettivamente sugli AC e sui WTP;
- implementa il *binding* con lo standard 802.11;
- supporta unicamente l'architettura Local MAC;
- per il protocollo DTLS, usa l'implementazione offerta da OpenSSL[3].
- è un codice multi-thread (fa uso delle primitive della libreria pthread)

Data la complessità del protocollo che implementa, la strutturazione del codice in più thread appare quasi una scelta obbligata [2]. Nei prossimi paragrafi (2.1.1, 2.1.2) viene illustrata la struttura di questi thread.

1 http://sourceforge.net/projects/open-capwap/
2 http://www.openwrt.org/
3 http://www.openssl.org/

Inoltre, nelle ultime versioni, è stata implementata un'interfaccia basata su socket attraverso la quale le applicazioni esterne possono comunicare con OpenCAPWAP per inviare e ricevere messaggi dai WTP. Questa interfaccia conferisce al codice una struttura modulare che permette di implementare agevolmente nuovi messaggi. Tale interfaccia sarà analizzata nel paragrafo 2.2.

L'ultimo paragrafo, il 2.2.3, introduce la struttura e le funzionalità offerte da Remote UCI [6], un modulo di OpenCAPWAP che è stato utilizzato nella prima fase di debugging (Capitolo 3) per eseguire stress test del sistema.

2.1.1 Thread dell'AC

Il demone AC si compone dei seguenti thread (Figura 9):

- **Ricezione pacchetti**

  L'AC usa un socket UDP per la ricezione dei pacchetti. La ricezione deve avvenire costantemente e indipendentemente dalle altre componenti dei codice: per questa ragione tale funzione deve essere implementata in un thread separato. Questo thread si occupa, inoltre, di trasferire i pacchetti ai thread che gestiscono i WTP.

- **Gestione dei WTP**

  Per ogni nuovo WTP autenticato, l'AC genera un thread. Il compito di questo thread è quello di gestire le richieste e le risposte relative al suo WTP.

- **Gestione dei timer**

  Data la complessità del protocollo, la gestione dei timer è demandata ad un thread dedicato. Questo thread funge da punto di riferimento per tutti gli altri thread che possono richiedere un timeout e, tramite un apposito segnale, invia la notifica della scadenza del timeout stesso.

- **Interfaccia con le applicazioni**

  Questo thread si interfaccia con le applicazioni esterne tramite un socket TCP (paragrafo 2.2).

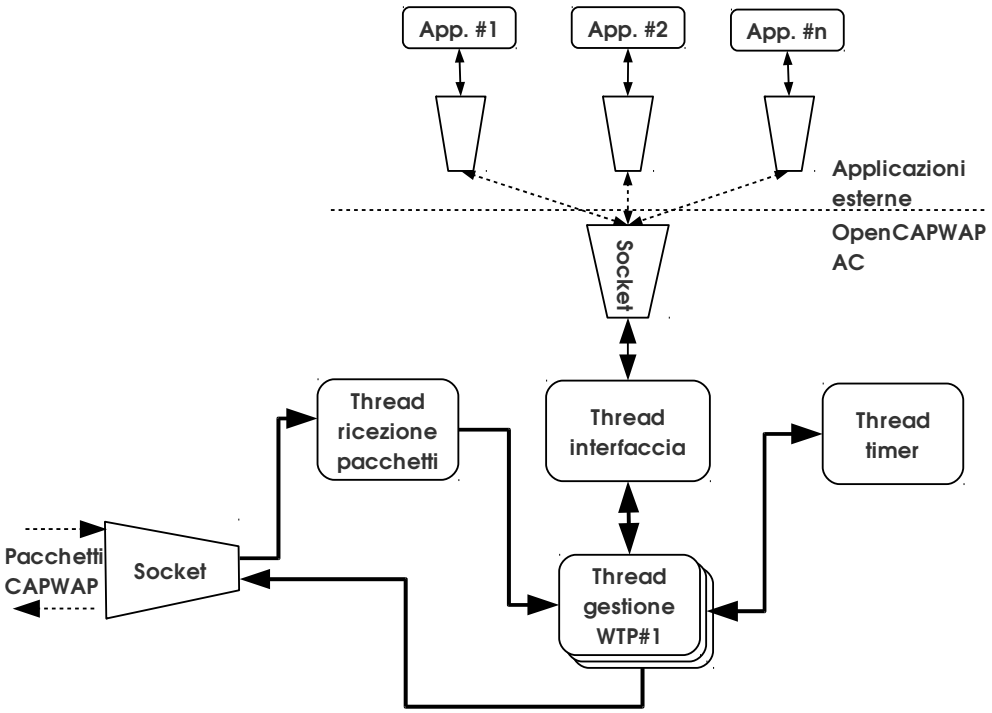

Figura 9: OpenCAPWAP - Thread dell'AC

2.1.2 Thread del WTP

Il demone WTP, più semplice rispetto all'AC, si compone dei seguenti thread (Figura 10):

- **Ricezione dei pacchetti**

  Il thread si occupa di ricevere e inviare pacchetti da e verso l'AC.

- **Thread principale**

  Coordina le varie funzionalità del WTP ed effettua le transizioni tra gli stati.

- **Ricezione frame 802.11**

  Questo thread recupera i frame del protocollo 802.11 provenienti dalle STA associate. Oltre a recuperare i frame, esso si occupa anche di gestire le statistiche riguardanti il traffico radio.

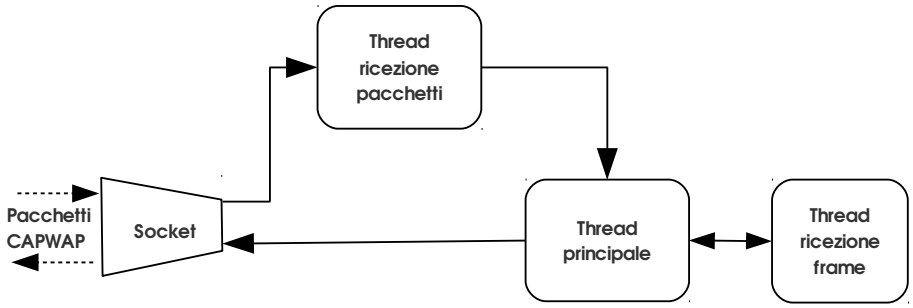

Figura 10: OpenCAPWAP - WTP Thread

2.2 Interfaccia dell'AC

Come accennato in precedenza, l'AC mette a disposizione un'interfaccia per consentire di implementare alcune funzioni del protocollo come moduli (ovvero applicazioni esterne). L'interfaccia è basata su socket di tipo *stream* (che utilizzano il protocollo TCP). In questo modo le applicazioni esterne possono essere sviluppate in un qualsiasi linguaggio di programmazione e non si è più legati al C. Di seguito, viene descritto il protocollo di comunicazione che le applicazioni esterne devono usare per colloquiare con l'AC.

2.2.1 Connessione

Il protocollo prevede che le applicazioni esterne si connettano sulla porta 1235, su cui è in ascolto l'AC. All'atto dell'apertura della connessione, l'AC invia all'applicazione un intero di 4 byte: se il valore è 1, vuol dire che la connessione può essere accettata. Un valore diverso da 1 indica che l'AC non può servire la richiesta; in particolare, il valore -1 specifica che la coda delle applicazioni è piena (l'AC gestisce un numero massimo di applicazioni esterne contemporaneamente).

2.2.2 Messaggi

Se la connessione va a buon fine, l'AC resta in ascolto di richieste. Le applicazioni hanno a disposizione tre tipi di messaggi per inviare richieste:

- **QUIT**: richiede la chiusura della connessione (Figura 11).
- **LIST**: richiede all'AC la lista dei WTP attualmente attivi. Il messaggio si compone di un singolo intero su 8 bit che corrisponde al codice del comando *LIST* (Figura 11). L'AC risponde con un messaggio contenente il numero dei WTP associati e le informazioni su ciascuno di essi (Figura 13).

- **CONFIGURATION UPDATE**: richiede all'AC di inviare un messaggio di *Configuration Update Request* ad un WTP. Come mostrato in Figura 12, è necessario specificare, oltre al campo CMD_TYPE (che sarà *CONFIGURATION UPDATE*), anche il campo MSG_ELEM (ad esempio *Vendor Specific Payload*), l'id del WTP ed eventualmente il *payload* che la richiesta di configurazione deve trasportare. L'eventuale risposta (Figura 14) porterà con se l'id del WTP da cui proviene, un codice per indicare l'esito dell'operazione di configurazione (*return code*) ed un eventuale *payload* contenente informazioni aggiuntive.

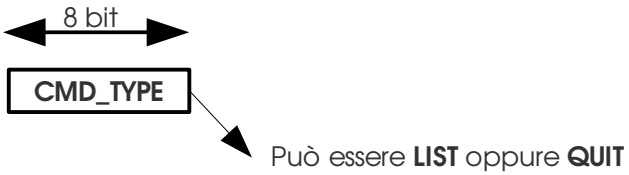

Figura 11: Interfaccia AC: Struttura dei messaggi QUIT e LIST

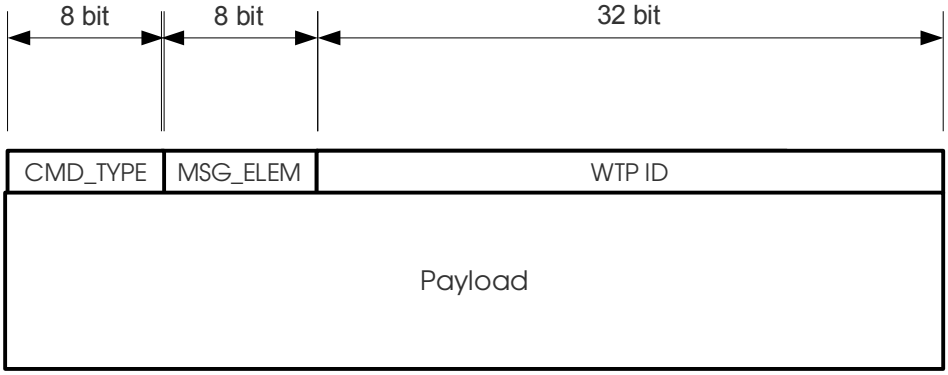

Figura 12: Interfaccia AC: Messaggio di Configuration Update

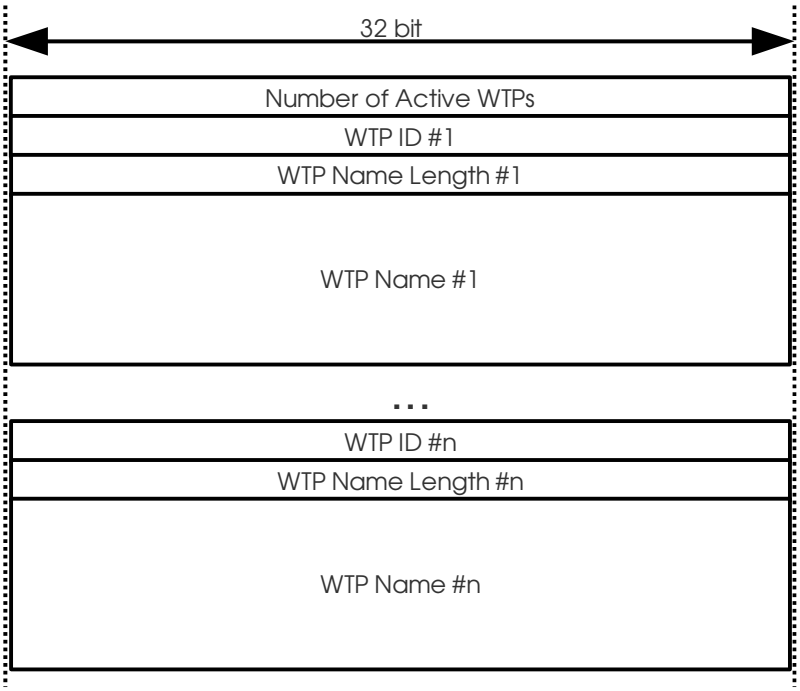

Figura 13: Interfaccia AC: Risposta al messaggio LIST

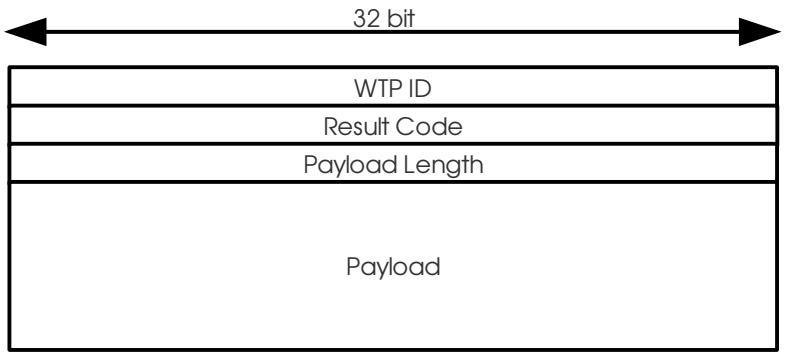

Figura 14: Interfaccia AC: Risposta al messaggio di Configuration Update

2.2.3 Un esempio di applicazione esterna: Remote UCI

Un esempio di applicazione basata su questa interfaccia è Remote UCI [6]. La distribuzione OpenWrt, su cui viene attualmente testato OpenCAPWAP lato WTP, offre un sistema di amministrazione centralizzato chiamato UCI. Remote UCI consente, tramite OpenCAPWAP, di configurare dinamicamente un WTP interagendo con UCI.

Il sistema ha due componenti:

- **Uci remoto:** comando lato AC, scritto in Python. Tramite l'interfaccia socket, può inviare *Configuration Update Request*. Consente di visualizzare e modificare la configurazione di uno o più WTP associati all'AC.
- **Server Uci**: server lato WTP. Quando il demone WTP riceve una *Configuration Update Request*, la gira al server Uci. Il server, dialogando con il sistema UCI, gestisce la richiesta e risponde al WTP. Ricevuta la risposta, il WTP la impacchetta in una *Configuration Update Response* e la inoltra all'AC.

Capitolo 3
Debugging di OpenCAPWAP

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
> -- ***Brian Kernighan***

Questo capitolo riporta i risultati della fase del tirocinio dedicata al debugging della release 0.91 di OpenCAPWAP. Il mio lavoro, in questa fase, ha portato alla produzione di alcune *patch*, contenenti *bugfix*, poi confluite nella release 0.92, rilasciata su *Sourceforge.org*.

Nel paragrafo 3.1 vengono descritti gli strumenti scelti per il debugging e le principali problematiche che essi aiutano ad affrontare; nel paragrafo 3.2 viene presentata la metodologia adottata durante il debugging e i risultati ottenuti.

3.1 Scatola degli attrezzi

Il primo basilare metodo per la ricerca degli errori è l'uso delle stampe di debugging: semplici funzioni printf(), nel caso del C, con le quali il programmatore può controllare lo stato del programma in determinati punti critici. Questo metodo risulta adeguato e rapido per programmi di piccole dimensioni caratterizzati da una logica applicativa semplice. Tuttavia, quando le dimensioni e la complessità del software oggetto del debugging aumentano, esso si dimostra inappropriato in quanto eccessivamente tedioso ed esposto, a sua volta, agli errori.

Nel caso di un software dalla logica applicativa complessa e dalle dimensioni estese come Open CAPWAP (~ 20.000 "*Lines of Code*" o LoC), appare evidente la necessità di fare ricorso a tecniche e strumenti più raffinati e potenti. Nei seguenti paragrafi descriverò i due strumenti principali che ho scelto di utilizzare per il tirocinio: Gdb e Valgrind.

3.1.1 Gdb: il debugger GNU

Il debugger è uno strumento che consente di ispezionare interattivamente i segmenti di *stack* e di dati globali di un processo. Con un debugger è quindi possibile analizzare lo stato di un programma (variabili locali/globali, parametri delle funzioni, etc...) in un qualunque momento della sua esecuzione.

Gdb[4] è il debugger a livello di codice sorgente[5] sviluppato nell'ambito del progetto GNU[6]; esso è in grado di operare con programmi scritti in diversi linguaggi (C, C++, ...) e offre un'interfaccia testuale completa.

Con Gdb è possibile:

- eseguire un programma stabilendo alcuni punti di interruzione chiamati *breakpoint*; la semantica del meccanismo prevede che, ad ogni *breakpoint*, l'esecuzione si interrompa;

- in corrispondenza dei punti di interruzione il programmatore può interagire con gdb usando una serie di comandi. E' possibile, ad esempio, controllare i valori delle variabili e i parametri attuali delle funzioni (avendo accesso allo *stack*, si ha anche accesso alla gerarchia delle chiamate ed è quindi possibile sapere in che punto del programma e da quale componente è stata chiamata una determinata funzione);

- è quindi possibile riprendere l'esecuzione istruzione per istruzione oppure fino al prossimo *breakpoint*.

Sebbene consenta di eseguire le operazioni di debugging molto rapidamente, la console testuale di gdb può risultare scomoda da utilizzare. Per questa ragione, sono state implementate numerose GUI che consentono il debugging visuale dei programmi. Alcuni esempi noti: *DDD*, *Insight* e *Kdbg*[7]. Quest'ultima GUI, di cui è riportata una schermata nella Figura 15, è realizzata usando le librerie QT[8] e offre un ambiente grafico completo e intuitivo per il debugging di applicazione di medie e grandi dimensioni. Per queste ragioni, per il tirocinio si è scelto di usare proprio Kdbg.

4 http://www.gnu.org/software/gdb/

5 In realtà gdb è in grado di lavorare anche a un livello più basso, ossia a livello di linguaggio macchina.

6 GNU – GNU is Not Unix: progetto mirato alla realizzazione di un sistema operativo Unix completamente libero (http://it.wikipedia.org/wiki/GNU)

7 http://www.kdbg.org/

8 QT: librerie grafiche su cui è basato l'ambiente desktop KDE

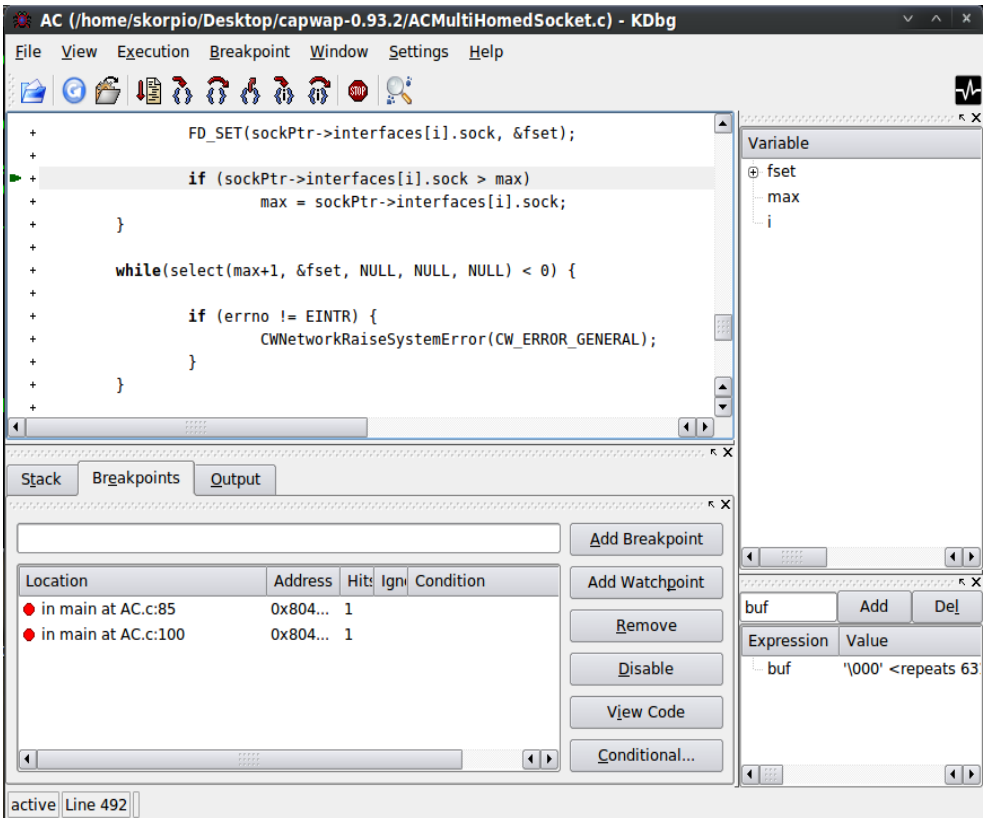

Figura 15: Schermata di Kdbg

Dietro le quinte: il meccanismo dei breakpoint

Spesso ci capita di usare delle tecnologie senza preoccuparci di studiarne i principi di funzionamento. Ed ecco che certi meccanismi di uso quotidiano sembrano scaturire da chissà quale alchimia. Lo scopo di questo paragrafo "dietro le quinte" è quello di illustrare come i debugger implementano il meccanismo dei breakpoint.

Va subito detto che esistono due strategie implementative: una software, l'altra hardware. Qui tratterò solo della prima che, sebbene meno performante, è sicuramente più versatile.

I breakpoint software sono realizzati dai debugger inserendo una speciale istruzione nel punto del programma in cui lo sviluppatore desidera avere un breakpoint. Nel caso della piattaforma Intel, l'istruzione è '*int 3*'. Essa non fa altro che richiamare un *exception handler* predisposto dal debugger [7]. L'inserimento di questa istruzione avviene, a grandi linee, a tempo di esecuzione, nel seguente modo:

- quando si inserisce un breakpoint ad una linea di codice, il debugger salva l'*opcode* presente a quella locazione e lo rimpiazza con *int 3*;
- quando l'esecuzione raggiunge l'istruzione *int 3*, il controllo passa al debugger;
- il debugger notifica all'utente che è stato raggiunto il breakpoint. A questo punto lo sviluppatore può analizzare lo stato del programma. Successivamente, richiederà al debugger di riprendere l'esecuzione;
- il debugger rimpiazza l'*opcode* di *int 3* con quello precedentemente salvato, in modo che l'istruzione originaria possa essere eseguita;
- dopo che l'istruzione originaria è stata eseguita, il debugger deve sostituirla nuovamente con l'*int 3*. Se questo passo non venisse fatto, il breakpoint si perderebbe (ovviamente ciò non vale per i breakpoint temporanei);
- il debugger riprende quindi l'esecuzione del programma.

Diversamente dal supporto hardware (il quale consente di inserire solo un numero limitato di punti di interruzione), questo meccanismo risulta molto versatile. I breakpoint software sono, infatti, completamente sotto il controllo del debugger - e, quindi, del programmatore - che può deciderne a piacimento posizione e numero.

3.1.2 Bug relativi alla gestione della memoria

Senza ombra di dubbio, una delle maggiori fonti di bug all'interno di programmi C è la gestione non corretta della memoria (sia essa statica o dinamica). E' singolare che molti sostengano, forse esagerando, che Java sia nato proprio per far fronte a questo tipo di errori. Errori da cui nessun programmatore C, per quanto bravo, è mai rimasto indenne.

Eppure, a prima vista, il meccanismo della memoria dinamica appare a dir poco banale. Esso è composto essenzialmente da due primitive: malloc() e free(). Ciascuna di essere prende un singolo argomento. Con la malloc() chiediamo l'allocazione di un blocco di memoria; poi, quando quel blocco non serve più, lo liberiamo usando la free(). Se non commettessimo mai errori, la gestione della memoria non costituirebbe alcun problema. Il punto è che, a causa della nostra natura umana, siamo condannati a commettere errori.

Un errore tipico si verifica quando allochiamo memoria che poi dimentichiamo di liberare. Questo errore determina una situazione detta "*memory leak*". Anche un *leak* di pochi byte, che a prima vista sembrano una quantità irrisoria, può avere effetti fatali. Se infatti il *leak* si verifica all'interno di un ciclo di un processo demone, presto o tardi la memoria, che è una risorsa finita, si esaurirà. Un *leak* può essere ovviamente causato da una malloc() a cui non corrisponde una free(), ma anche dall'accidentale cancellazione, perdita o

sovrascrittura di un puntatore ad una zona di memoria precedentemente allocata.

Come se non bastasse, oltre ai *memory leak*, i programmatori C devono fare i conti con i problemi causati dalla corruzione della memoria. Per corruzione della memoria si intende una scrittura non opportunamente controllata del contenuto di una zona di memoria, sia essa statica oppure allocata dinamicamente dal programmatore (nell'*heap*) o dal compilatore (sullo *stack*). Il "*buffer overrun*", che si verifica quando si scrive oltre i confini di un blocco di memoria, è una causa frequente di corruzione della memoria. Un tipo molto insidioso di *buffer overrun* è l'OBOE (*Off-By-One Error*). Questo errore, piuttosto comune tra i principianti, si verifica quando si scrive un byte oltre i limiti di una zona di memoria. Ecco un tipico esempio che ha a che fare con la copia di stringhe:

```
char *str2 = malloc(strlen(str1));
strcpy(str2, str1);
```

Con questo codice, il programmatore voleva copiare il contenuto della stringa *str1* in una nuova stringa *str2*. L'errore è quello di allocare per *str2* un byte in meno di quanto serva: questo avviene perché *strlen*() restituisce la lunghezza di una stringa senza considerare il carattere di terminazione ('\0').

Gli errori di gestione della memoria possono avere effetti inaspettati e, come detto prima, addirittura nefasti. Se un processo continua ininterrottamente a "mangiare" memoria, il sistema operativo farà sempre più fatica e le prestazioni degraderanno: a questo punto le conseguenze si ripercuoteranno anche su tutti gli altri incolpevoli processi. Sono ancora peggiori le situazioni in cui la memoria si corrompe: il processo potrebbe iniziare a manifestare casualmente comportamenti inattesi o *crash* spuri sempre in momenti e punti diversi del codice. Non bisogna poi dimenticare che, dove c'è un bug, c'è un attaccante pronto a sfruttarlo a suo vantaggio.

Gli errori di gestione della memoria sono, come abbiamo visto, subdoli e possono restare nascosti un tempo indeterminato prima di manifestare i sintomi iniziali; sintomi talvolta ingannevoli che rendono molto difficile identificare e correggere la causa reale del problema. Per questo motivo, nel corso degli anni, sono stati sviluppati strumenti sempre più efficaci per aiutare il programmatore in questo difficile compito. I più noti, in ambito Linux, sono MEMWATCH, Electric Fence e Valgrind. Per il debugging di Open CAPWAP, la scelta è ricaduta su Valgrind: infatti, nonostante sia il più legato alla piattaforma hardware/software (funziona solo con Linux x86), esso è anche il più completo e sofisticato.

3.1.3 *Valgrind: Memory-Management Debugger*

Valgrind[9] [10] è stato inizialmente progettato e scritto da Julian Seward che lo ha rilasciato alla comunità sotto licenza GPL. Successivamente, grazie al consueto modello di sviluppo "a più mani" tipico del software libero, Valgrind si è evoluto trasformandosi in un *framework* avanzato per la creazione di strumenti di analisi dinamica del codice [8]. La sua versatilità e affidabilità sono confermate dal numero sempre crescente di progetti software che ne fanno uso. Tra questi basti citare Firefox, OpenOffice, Samba, PHP e Unreal Tournament[11].

Valgrind è in grado di individuare una vasta gamma di problemi legati alla gestione della memoria, tra i quali figurano [9]:

- uso di memoria non inizializzata;
- lettura e scrittura di memoria che è stata liberata;
- lettura e scrittura di memoria fuori dai confini dei blocchi effettivamente allocati;
- lettura e scrittura errate sullo *stack*;
- *memory leak*;
- passaggio di puntatori a memoria non inizializzata o non indirizzabile;
- malloc() o new[12] a cui non corrispondono free() o delete.

Dietro le quinte: architettura di Valgrind

Come fatto per i breakpoint, anche qui descriviamo brevemente cosa avviene dietro le quinte di Valgrind per capirne il funzionamento.

Valgrind non è altro che un emulatore di processore (piuttosto che una macchina virtuale) che usa la tecnica della compilazione dinamica o compilazione JIT - just-in-time. Prima di essere eseguito, il programma viene tradotto in una forma intermedia, più semplice, chiamata *Intermediate Representation* (IR): una forma neutra, indipendente dal processore, basata su istruzioni SSA[13]. Quando il codice è in questa forma, su di esso vengono eseguiti uno o più strumenti: gli strumenti non fanno altro che analizzare e modificare il codice aggiungendo, ad esempio, codice di controllo. Alla fine, il codice modificato viene ritradotto nel linguaggio macchina della CPU ed è pronto per essere eseguito[10]. La Figura 16 mostra questa architettura.

9 http://valgrind.org/

10 Il nome Valgrind è presto a prestito da una divinità della mitologia nordica.

11 http://valgrind.org/gallery/users.html

12 L'istruzione 'new' è usata nel linguaggio C++ per allocare memoria all'atto della creazione di nuovi oggetti. La memoria viene poi liberata con il metodo 'delete'.

13 SSA – *Static Single Assignment*: forma di rappresentazione intermedia in cui ogni variabile è assegnata solo una volta. Grazie a questa proprietà, questa rappresentazione consente ai compilatori di migliorare e velocizzare le procedure di ottimizzazione del codice.

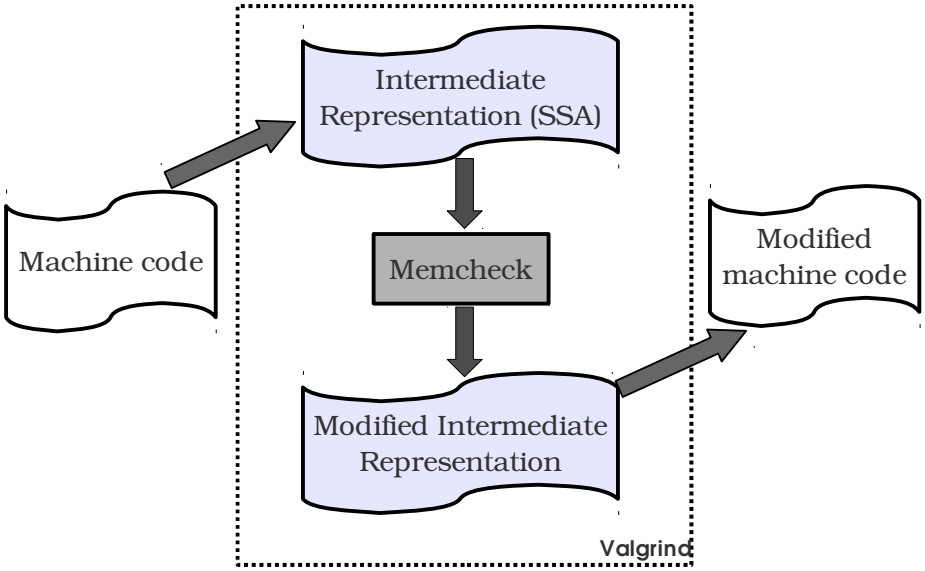

Figura 16: Valgrind Architecture

Questa organizzazione modulare ha favorito lo sviluppo di numerosi strumenti di debugging e profiling[14]. Il più noto è il Memcheck, usato per rilevare errori relativi alla gestione della memoria. Memcheck aggiunge un prologo prima di (quasi) tutte le istruzioni del programma: il compito di questo prologo è quello di controllare la validità e l'indirizzabilità delle zone di memoria interessate dall'istruzione. Inoltre, Memcheck rimpiazza il gestore di memoria standard della libreria C con una sua implementazione. Tale implementazione aggiunge opportuni controlli attorno ai blocchi di memoria allocati, in modo da rilevare con esattezza gli errori di *buffer overrun* e OBOE.

Tutto ciò ha ovviamente un prezzo notevole in termini di prestazioni: si stima che, in queste condizioni, un programma giri dalle cinque alle venti volte più lentamente ed usi una quantità considerevolmente maggiore di memoria[10]. A patto di avere a disposizione una macchina sufficientemente performante, questo non dovrebbe costituire un reale problema.

3.2 Ricerca e correzione degli errori

Dopo aver analizzato gli strumenti (GDB e Valgrind), esponiamo nei prossimi paragrafi la metodologia adottata per il debugging e i risultati ottenuti.

14 Un elenco completo dei principali strumenti può essere trovato al seguente link http://valgrind.org/info/tools.html

3.2.1 Metodologia

Nell'approccio al debugging di software di medie e grandi dimensioni può essere utile fare riferimento ad una metodologia sistematica che aiuti a gestire la complessità del processo di ricerca e correzione degli errori. Molto è stato scritto in merito. Tuttavia, prima di procedere, tengo a sottolineare quale deve essere, a mio avviso, l'approccio del programmatore verso queste metodologie. Un approccio scorretto, infatti, può arrecare più danni che benefici[15].

Per approccio scorretto intendo un approccio di tipo dogmatico: lo sviluppatore segue pedissequamente tutte le fasi prescritte senza la necessaria elasticità e comprensione. Per approccio corretto intendo, invece, l'approccio del programmatore che, compreso lo spirito della metodologia, ne segue le prescrizioni considerandole linee guida (non dogmi) e li adatta con elasticità ai casi specifici.

Dopo questa doverosa considerazione, passo a esporre le linee guida principali che ho adottato, sintesi di quanto esposto in [9][11][12].

1. Studiare il software

Se non si conosce il software, prima di tutto è necessario studiarne, a grandi linee, la struttura e il funzionamento.

2. Far fallire il software

Riprodurre i bug; semplificare, se possibile, il caso da controllare.

3. Analizzare i log

L'analisi dei log è fondamentale. Se l'errore non appare nei log, è necessario, dopo la correzione, aggiornare il sistema.

4. Scegliere gli strumenti adeguati e comprenderne il funzionamento

Come detto in precedenza, il processo di debugging è complesso e può richiedere molto tempo. Per avere successo, è quindi necessario affrontarlo con gli strumenti adeguati. Servirà sicuramente un debugger generico e, nel caso di linguaggi come il C, anche un debugger della memoria. Scelti gli strumenti, bisogna poi studiarne le funzionalità e capire come esse sono implementate.

5. Isolare il componente in cui è presente il bug

Quando il software è complesso, bisogna eliminare per gradi tutti qui componenti che non sono interessati dal bug.

15 A riguardo, si pensi al fallimento del RUP – Rational Unified Process – nell'Ingegneria del Software.

6. Individuare la causa reale del bug

Distinguere i fatti dalla loro interpretazione ed evitare di saltare a conclusioni affrettate.

7. Correggere il bug

Progettare e implementare una correzione per il bug. Sviluppare una modifica alla volta.

8. Testare ripetutamente

Testare accuratamente la correzione; il bug scompare sempre? La correzione ha introdotto delle regressioni nel codice?

9. Se il bug non è stato corretto, allora è ancora lì

Alcune volte, modificando del codice che non sembra avere nulla a che fare col problema, il bug scompare. Prima di considerare il bug risolto, bisogna riuscire a capire in che modo la modifica introdotta abbia corretto l'errore.

Altre volte, un bug sembra scomparire da solo. Non illudersi: i bug, a dispetto del nome, non sono esserini che si annidano nel codice e che possono decidere di migrare altrove autonomamente. La scomparsa di un bug vuol semplicemente dire che, per qualche ragione, non siamo più in grado di riprodurlo. Abbiamo quindi fatto un passo indietro; dobbiamo analizzare di nuovo tutte le possibili situazioni per ricreare le condizioni in cui il bug possa nuovamente manifestarsi.

10. Se possibile, coprire il bug con un test di regressione

In stadi successivi dello sviluppo, un bug risolto potrebbe essere reintrodotto (regressione). In questi casi, è utili predisporre dei test automatizzati che catturino i bug già risolti ed eseguirli periodicamente per rilevare eventuali regressioni.

3.2.2 Come sono stati condotti i test?

Dalle segnalazioni raccolte, è risultato subito evidente che il codice di Open CAPWAP avesse dei bug prevalentemente nella gestione della memoria. Per questa ragione, si è deciso di condurre l'analisi usando uno *stress test*. Lo *stress test* predisposto consiste in uno script di shell *bash* che utilizza intensivamente il comando Remote UCI (paragrafo 2.2.3) per inviare *Configuration Update Request* ad un WTP.

Inizialmente, lo *stress test* è stato lanciato eseguendo l'AC sotto il controllo di Valgrind. Successivamente, lo stess test è stato ripetuto eseguendo, questa volta, il WTP sotto il controllo di Valgrind. Si è quindi proceduto a raccogliere e

catalogare le varie segnalazioni di errore, analizzando il codice a cui esse facevano riferimento. A ciascun bug scoperto è stato assegnato un codice identificativo.

Gli errori riscontrati sono stati suddivisi quattro tipologie:

- **ML -** *Memory Leak*: di cui si è ampiamente ampiamente parlato nel paragrafo 3.1.2.
- **UMR -** *Uninitialized Memory Read*: letture a zone di memoria non inizializzata.
- **TRL -** *Thread Resource Leak*: errori nella gestione delle risorse dei thread.
- **LE -** *Logic Error*: errori nella logica applicativa.

Il prossimo paragrafo mostra una lista sintetica dei bug e alcune statistiche sulle loro occorrenze nel codice.

3.2.3 Statistiche ed elenco degli errori

La tabella di Figura 19 elenca i bug riscontrati, indicando per ciascuno: un codice (da cui è possibile dedurre la tipologia di bug), la data di scoperta, il componente in cui è stato riscontrato (CW indica che il bug è comune sia all'AC che al WTP) e la *patch* che lo corregge. Una descrizione dettagliata dei bug è riportata in appendice a questo lavoro.

Il grafico in Figura 17 mostra le occorrenze dei bug suddivisi per tipologia. Esso, nel caso ve ne fosse ancora bisogno dopo quarant'anni, non fa altro che confermare una triste verità: la gestione della memoria è il tallone di Achille del C.

Il grafico in Figura 18 mostra, invece, le occorrenze dei bug nelle varie componenti di OpenCAPWAP. Non sorprende constatare come la maggior parte dei bug fosse annidata nel codice del WTP: la maggior parte delle segnalazioni di errore iniziali facevano infatti riferimento ai WTP.

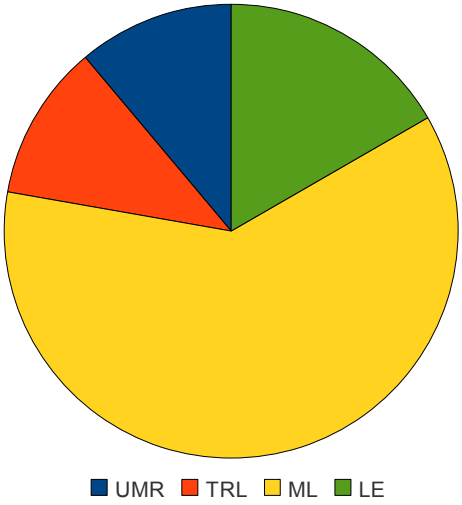

Figura 17: OpenCAPWAP 0.91 – Grafico delle occorrenze dei bug

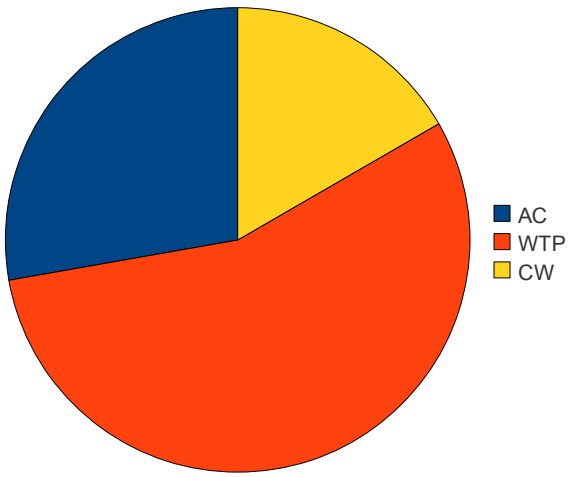

Figura 18: OpenCAPWAP 0.91 - Distribuzione dei bug per componenti

ID	Data	Componente	Patch
UMR02	12/10/09	AC	--
TRL01	14/10/09	AC	capwap-TRL01-LE01.patch
LE01	14/10/09	AC	capwap-TRL01-LE01.patch
ML01	15/10/09	WTP	capwap-ML01.patch
TRL02	15/10/09	CW	capwap-TRL02.patch
ML02	16/10/09	WTP	capwap-ML02-06.patch
ML03	16/10/09	WTP	capwap-ML02-06.patch
ML04	16/10/09	WTP	capwap-ML02-06.patch
ML05	16/10/09	WTP	capwap-ML02-06.patch
ML06	16/10/09	WTP	capwap-ML02-06.patch
LE02	19/10/09	CW	capwap-GENERAL.patch
ML07	19/10/09	WTP	capwap-GENERAL.patch
ML08	20/10/09	WTP	capwap-GENERAL.patch
ML09	20/10/09	WTP	capwap-GENERAL.patch
UMR02	20/10/09	WTP	capwap-GENERAL.patch
LE03	22/10/09	CW	capwap-GENERAL.patch
ML11	22/10/09	AC	capwap-GENERAL.patch
ML12	22/10/09	AC	capwap-GENERAL.patch
UMR03	24/10/09	AC	capwap-GENERAL.patch
LE04	20/11/09	AC	capwap-LE04.patch

Figura 19: Elenco bug

Capitolo 4
Sistema di aggiornamento dei WTP

"Frustra fit per plura quod fieri potest per pauciora."
(È inutile fare con più ciò che si può fare con meno.)
*-- **William di Ockham***

Questo capitolo descrive il sistema di aggiornamento dei WTP che ho progettato e sviluppato per OpenCAPWAP. Questa fase dello sviluppo ha portato al rilascio su *Sourceforge.org* della versione 0.93.2.

4.1 Obiettivi

Il sistema di aggiornamento è stato progettato per essere il più generale possibile e reagire automaticamente ai malfunzionamenti. In particolare, ecco gli obiettivi prefissati:

- **indipendenza**: il sistema non deve dipendere dalla sottostante distribuzione di Unix/Linux;
- **generalità e flessibilità**: l'aggiornamento deve essere possibile anche qualora la struttura del WTP cambiasse radicalmente;
- **resistenza ai malfunzionamenti**: se durante la fase di aggiornamento avvenisse qualche errore, il sistema dovrebbe essere in grado di ripristinare una versione funzionante del WTP.

4.1.1 Note e considerazioni preliminari

A causa del primo requisito di indipendenza, è evidente che la procedura di aggiornamento del demone WTP deve essere indipendente dalla procedura di aggiornamento del *firmware* del WTP (inteso come intero SO) che è invece descritta dalle specifiche del protocollo CAPWAP [3]. Da queste considerazioni consegue anche che, per l'aggiornamento, non è possibile basarsi sui sistemi di pacchetti specifici delle varie distribuzioni. E' stato quindi necessario introdurre un formato ad-hoc per i pacchetti di aggiornamento chiamato CUP, che verrà descritto nel seguito.

Prima di procedere nella descrizione è opportuno definire brevemente cosa si intende per **aggiornamento del WTP**. Come sarà evidente dall'analisi del formato dei pacchetti CUP, un aggiornamento può portare con se tutto ciò che è contenuto nella directory di lavoro del demone WTP. Nell'implementazione attuale (0.93.2), tale directory coincide con la directory che contiene i binari, la configurazione e le chiavi per la connessione DTLS. Invece i log, che fino alla versione 0.92 erano contenuti nella stessa directory, sono stati trasferiti in */var/log*.

4.1.2 Versioni di OpenCAPWAP

Durante la fase di progettazione, ho dovuto stabilire il sistema con il quale vengono assegnati i numeri di versione ad OpenCAPWAP. Il numero di versione è specificato da tre interi separati da punto, ciascuno col seguente significato (Figura 20):

- **MAJOR NUMBER**: usato per indicare cambiamenti rilevanti rispetto alle versioni precedenti;
- **MINOR NUMBER**: usato per correzioni di bug e aggiunta di funzionalità minori;
- **REVISION NUMBER**: correzioni di bug minori o versioni di sviluppo intermedie non necessariamente rilasciate al pubblico.

Figura 20: Numeri di versione di OpenCAPWAP

Il numero di versione può essere modificato nell'header file WUM.h.

4.2 Architettura del sistema

Il sistema di aggiornamento aggiunge due componenti ausiliari al codice di OpenCAPWAP:

- **WUM** – *WTP Update Manager*: comando lato AC che consente di inviare richieste di aggiornamento ai WTP;
- **WUA** – *WTP Update Agent*: processo lato WTP che si occupa di portare a termine la procedura di aggiornamento.

La procedura di aggiornamento è avviata lato AC a seguito di una richiesta effettuata dall'operatore tramite il comando WUM. WUM, tramite dei messaggi aggiunti al protocollo, trasferisce al WTP l'aggiornamento sotto forma di un archivio compresso chiamato **CUP** – *CAPWAP Update Package.* Ricevuto correttamente l'aggiornamento, il WTP termina e passa il controllo al WUA che, come detto, si occupa di applicare l'aggiornamento e, al termine della procedura, di avviare la nuova versione del WTP.

Nei prossimi paragrafi vengono descritte, in dettaglio, le componenti del sistema a cominciare dal formato dei pacchetti CUP.

4.2.1 CAPWAP Update Package

I pacchetti CUP sono archivi compressi con gzip[16] che fa uso dell'algoritmo di compressione DEFLATE[17]: algoritmo che unisce LZ77 e la Codifica di Huffman. Il contenuto dell'archivio è il seguente:

- **update.cud**
  - il CUD (*CAPWAP Update Descriptor*), un file di testo ASCII che descrive l'aggiornamento
- **WTP/**
  - directory che contiene i nuovi file (binari, file di configurazione, sotto-directory, ...)
- **scripts/**
  - directory che contiene eventuali script della shell di pre e post aggiornamento

Non c'è molto da aggiungere sulle directory WTP e scripts; è invece necessario approfondire la struttura del descrittore di aggiornamento.

CUD – CAPWAP Update Descriptor

Il CUD è un file di testo ASCII che descrive l'aggiornamento. Il suo formato, molto semplice, ha la struttura **opzione:valore**. In realtà, in una prima fase del disegno, si era pensato di ricorrere ad un formato XML. Successivamente, però, considerando (1) l'esiguità delle informazioni che questo file deve portare e (2) il costo che ha l'utilizzo di un *parser* XML su macchine dalle capacità limitate come quelle su cui girano i WTP, si è optato per l'utilizzo di un file ASCII in un formato molto più semplice e agevole da gestire.

16 http://it.wikipedia.org/wiki/Gzip

17 L'algoritmo fu previsto come sostituito di LZW, coperto da brevetti e quindi non utilizzabile all'interno del progetto GNU.

Le opzioni attualmente previste sono:

- **Version**: specifica la versione contenuta nell'aggiornamento (*MAJOR.MINOR.REVISION*);
- **PreUpdate**: script da eseguire prima che l'aggiornamento inizi (*pathname* relativo alla directory *scripts*);
- **PostUpdate**: script da eseguire al termine dell'aggiornamento (*pathname* relativo alla directory *scripts*).

I nomi delle opzioni sono insensibili alle maiuscole. L'unica opzione obbligatoria è '*Version*'.

Gli script di pre e post aggiornamento sono forniti per consentire di personalizzare il più possibile la procedura di aggiornamento. Ad esempio, possono essere usati per convertire i file di configurazione esistenti in un nuovo formato. Gli script hanno accesso a due variabili di ambiente: *$WTP_DIR* e *$CUP_DIR*. La prima contiene il *pathname* della directory di lavoro del WTP; la seconda il *pathname* della directory temporanea in cui è stato estratto il CUP. In Figura 21 riportato un esempio di CUD.

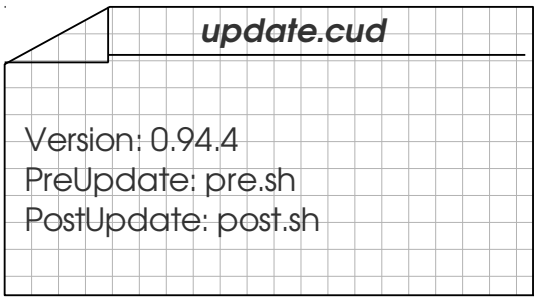

Figura 21: Esempio di update.cud

4.2.2 Stati e messaggi di aggiornamento

Il sistema di aggiornamento dei WTP mantiene un suo stato interno, indipendente dallo stato del WTP. Gli stati previsti sono tre:

- **WAIT:**

  il WTP è in attesa di richieste di aggiornamento.

- **BUSY**:

  il WTP ha accettato una richiesta di aggiornamento ed è impegnato a ricevere i frammenti del CUP relativo;

- **READY:**

  il WTP ha ricevuto con successo tutti i frammenti del CUP ed è pronto per eseguite il *WTP Update Agent*.

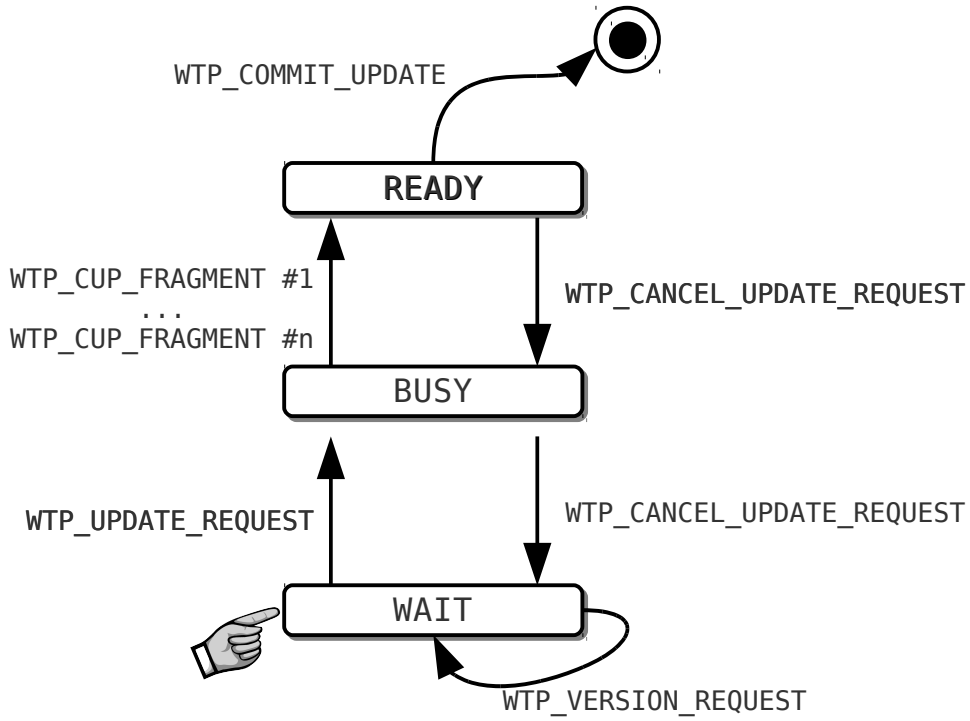

Figura 22: Stati del sistema di aggiornamento dei WTP

In accordo ai messaggi ricevuti, il sistema transita da uno stato ad un altro, come mostrato dall'automa a stati finiti in Figura 22: lo stato iniziale è WAIT e ciascuno spigolo dell'automa è etichettato con il messaggio che provoca la transizione di stato.

I messaggi aggiunti al protocollo CAPWAP da questo sistema di aggiornamento sono dieci:

- **WTP_VERSION_REQUEST**

  L'AC chiede al WTP di indicare la sua versione; il WTP risponde con un messaggio di tipo **WTP_VERSION_RESPONSE**.

- **WTP_UPDATE_REQUEST**

  L'AC chiede al WTP se può eseguire un aggiornamento. Nella richiesta vengono specificati la dimensione e la versione dell'aggiornamento. Il WTP risponde con una **WTP_UPDATE_RESPONSE**, il cui *Return Code* indica se l'aggiornamento viene accetto. In caso affermativo, il WTP apre un file temporaneo che sarà usato per ricevere il CUP e transita nello stato BUSY, in attesa di ricevere i dati che costituiscono l'aggiornamento.

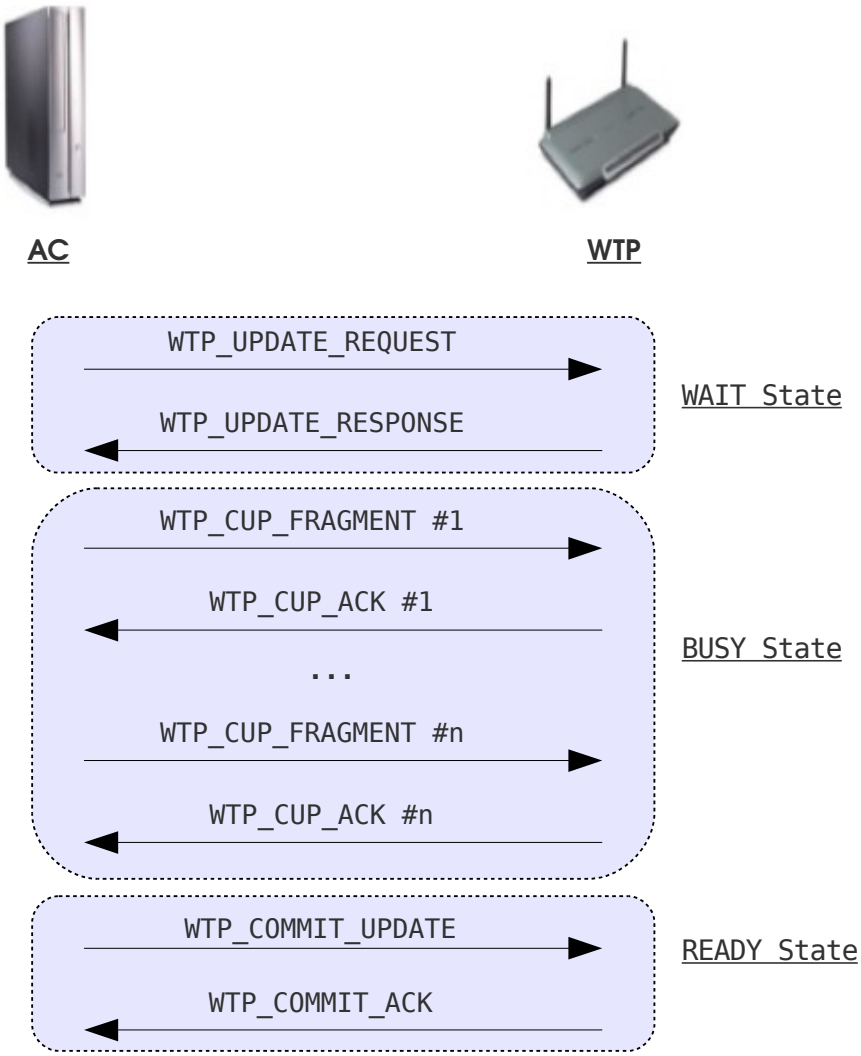

Figura 23: Scambio di messaggi tra AC e WTP durante una procedura di aggiornamento

- **WTP_CUP_FRAGMENT**

  Questo messaggio è usato per consegnare un frammento del CUP al WTP. Il CUP viene suddiviso in frammenti prima di essere inviato, e ciascun frammento ha dimensione 4000 byte, eccetto l'ultimo che può avere una dimensione inferiore. Il WTP conferma il frammento ricevuto inviando un **WTP_CUP_ACK**. Quando tutti i frammenti sono stati ricevuti, il sistema di aggiornamento del WTP passa nello stato READY.

- **WTP_COMMIT_UPDATE**

  L'AC richiede al WTP di terminare e di avviare la procedura di aggiornamento. Il WTP, dopo aver risposto positivamente con un WTP_COMMIT_ACK, termina e avvia la procedura di aggiornamento.

- **WTP_CANCEL_UPDATE_REQUEST**

  L'AC chiede al WTP di annullare un aggiornamento in corso. Questo messaggio viene accettato se il WTP è nello stato READY oppure BUSY, prima che sia stato inviato un WTP_COMMIT_UPDATE. Accettato il messaggio, il WTP cancella i file temporanei, torna nello stato WAIT e risponde con una **WTP_CANCEL_UPDATE_RESPONSE**.

I dieci messaggi sono raggruppati come *Configuration Update Request/Response* di tipo *Vendor Specific Payload*: questo tipo di messaggi è stato introdotto in CAPWAP per consentire ai produttori di estendere il protocollo con nuove funzionalità. La Figura 23 mostra uno scambio di messaggi tra AC e WTP durante una procedura di aggiornamento andata a buon fine.

4.2.3 WUM – WTP Update Manager

WUM è il comando testuale usato per interagire lato AC col sistema di aggiornamento. Scritto in C, questo comando funziona interagendo con l'interfaccia basata su socket TCP esposta nel paragrafo 2.2. Nel seguito vengono brevemente descritte le operazioni principali che è possibile eseguire con wum.

Ottenere la lista dei WTP associati

Di default, wum cerca di connettersi all'AC sulla porta locale 1235. Questo comportamento può essere modificato usando le opzioni *-a <indirizzo>* e *-p <numero di porta>*. Per ottenere una lista di tutti i WTP attivi è possibile usare l'opzione '*-c*' specificando il comando '*wtps*' ('c' sta per command e '*wtps*' è solo uno dei comandi supportati). Un esempio è mostrato in Figura 24.

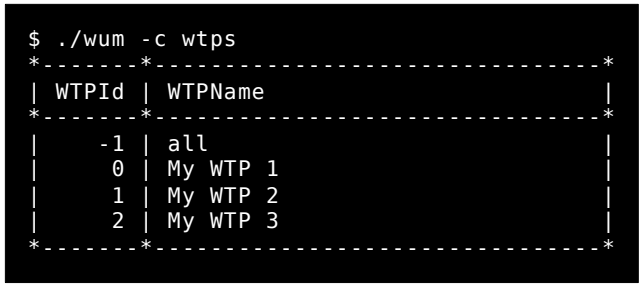

Figura 24: WUM: lista dei WTP associati

Controllare la versione dei WTP

Per controllare la versione dei WTP è possibile usare il comando '*version*'. Per indicare quali WTP interessano, si deve passare l'opzione *-w* (seguita dalla lista, separata da virgole, degli id dei WTP) oppure l'opzione *-n* (seguita dalla lista, separata da virgole, dei nomi del WTP). Usando -1 oppure 'all' è possibile specificare tutti i WTP associati. Di seguito alcuni esempi:

- wum -c version -w 0,2
- wum -c version -n 'My WTP 1','My WTP 3'

Inviare un aggiornamento

Per inviare un aggiornamento a un insieme di WTP bisogna usare il comando '*update*'. Questo comando, in aggiunta alla lista dei WTP coinvolti dall'aggiornamento, richiede che venga specificato un file CUP con l'opzione -f. Di seguito un esempio di utilizzo:

- wum -c update -n 'all' -f capwap0.93.4.cup

Annullare un aggiornamento

In caso di problemi durante l'invio del CUP, il sistema di aggiornamento potrebbe restare nello stato *READY* o *BUSY*. In questo caso, il WTP non accetterà ulteriori richieste di aggiornamento. In questo scenario, per poter riportare il sistema ad uno stato consistente e ritentare l'aggiornamento è necessario inviare una *WTP_CANCEL_UPDATE_REQUEST*. A questo scopo è possibile utilizzare il comando '*cancel*' di wum.

4.2.4 WUA – WTP Update Agent

Quando il WTP riceve un *WTP_COMMIT_UPDATE*, se si trova nello stato corretto (ossia *READY*), risponde con un *WTP_COMMIT_ACK*, termina la sua esecuzione e avvia il WTP Update Agent. Il compito di questo componente è quello di portare a termine una sessione di aggiornamento.

La sessione di aggiornamento è divisa in 3 stage:

- **STAGE 1**: *Preparazione*
  - estrazione del CUP in una directory temporanea (/tmp/cup.unpack)
  - salvataggio della directory di lavoro del WTP corrente (/tmp/cup.backup)
- **STAGE 2**: *Installazione dell'aggiornamento*
  - esecuzione dello script di pre-aggiornamento
  - copia dei nuovi file aggiornati (/tmp/cup.unpack/WTP/*) nella directory di lavoro del WTP
  - esecuzione dello script di post-aggiornamento
- **STAGE FINALE**: *Pulizia*
  - avvio del WTP aggiornato
  - pulizia dei file temporanei e terminazione del WUA

Se avvengono degli errori durante il primo stage, il WUA non fa altro che eliminare le directory temporanee e riavviare il vecchio WTP. Se invece avvengono errori durante gli stage successivi, il WUA ripristina la vecchia versione del WTP usando la copia salvata eseguito durante il primo stage. Quindi, rimuove i file temporanei e riavvia una versione funzionante del WTP.

Il WUA mantiene un log delle sessioni di aggiornamento nel file */var/log/wua.log*.

4.3 Considerazioni finali sul downgrade

L'implementazione attuale del sistema di aggiornamento non supporta esplicitamente il *downgrade*. Se l'utente prova ad inviare un pacchetto CUP con una versione inferiore a quella già installata, il WTP rifiuta la richiesta. Questo controllo è stato previsto per evitare che gli amministratori potessero inavvertitamente eseguire un *downgrade* non desiderato.

Per supportare, in futuro, la possibilità di effettuare un *downgrade,* possono essere percorse due strade:

- aggiungere una coppia di messaggi specifici dedicata al *downgrade*, *WTP_DOWNGRADE_REQUEST* e *WTP_DOWNGRADE_RESPONSE*;
- aggiungere un *flag* al messaggio di *WTP_UPDATE_REQUEST* per indicare che si desidera saltare il controllo di versione;

Di queste due strade, la seconda è meno invasiva in quanto richiede solo piccole modifiche al codice già prodotto.

Capitolo 5
Collaudo e conclusioni

"[...] se non v'è dispiaciuta affatto [la storia], vogliatene bene a chi l'ha scritta, e anche un pochino a chi l'ha raccomodata. Ma se in vece fossimo riusciti ad annoiarvi, credete che non s'è fatto apposta."
*-- **Alessandro Manzoni***

In questo capitolo viene descritta la fase finale del tirocinio dedicata al collaudo del sistema di aggiornamento dei WTP. La versione di OpenCAPWAP prodotta alla fine di questa fase, la 0.93.2, è stata rilasciata pubblicamente su *Sourceforge.org*. Alla fine di questo capitolo sono esposte le conclusioni sul lavoro e i possibili sviluppi futuri del progetto OpenCAPWAP.

5.1 Collaudo

La fase di collaudo e test è stata condotta presso il Consorzio CASPUR in un ambiente simulato. Gli apparati utilizzati come WTP sono degli *alix3d1*: si tratta di sistemi hardware completi ma di dimensioni minimali, particolarmente adatti per essere usati come Access Point. La scelta di collaudare il sistema su questi dispositivi è motivata dalla possibile introduzione delle funzionalità di aggiornamento all'interno della rete wireless *Provincia Wi-Fi*[18]. Tale rete, basata proprio su questi sistemi, mira ad offrire connettività wireless gratuita ad Internet nei principali luoghi pubblici della Provincia di Roma.

Gli *alix3d1* - di cui è mostrato un esemplare nelle figure 25, 26 - sono dotati di:

- un processore AMD Geodie a basso consumo da 433 Mhz;
- 128 MB di memoria RAM;
- un disco fisso costituito da una memoria Compact Flash;
- una scheda Ethernet (802.3);
- una scheda wireless (802.11) con connettori per antenna esterna.

18 http://www.provincia.roma.it/percorsitematici/innovazione-tecnologica/progetti/4035

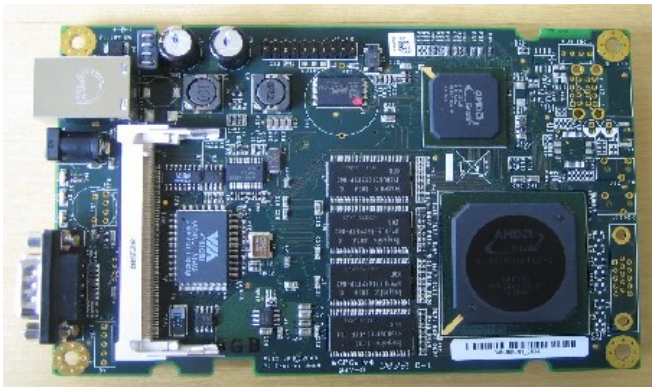

Figura 25: Scheda Alix3d1

Figura 26: Alix3d1 assemblato

Per quanto riguarda la configurazione software, sui WTP è stata installata la distribuzione di Linux *OpenWRT*, versione 8.09.2, nome in codice kamikaze[19]. Poiché questa distribuzione fa uso di uclicb[20], una versione ridotta delle librerie C GNU per sistemi *embedded*, è stato necessario compilare i moduli software WTP e WUA utilizzando un ambiente di sviluppo appositamente prelevato dal sito di OpenWRT.

19 http://downloads.openwrt.org/kamikaze/
20 http://www.uclibc.org/

5.1.1 *Batteria di test*

I test sono stati svolti utilizzando utilizzando come WTP quattro dispositivi *alix3d1* configurati nel modo esposto precedentemente ed equipaggiati con la versione 0.93.2 di OpenCAPWAP. Per automatizzare la ripetizione della batteria di test (Figura 27) sui quattro WTP, è stato progettato uno script *bash* eseguito la AC. Al termine di ogni comando della batteria, tale *script* verifica se l'esito corrisponde a quanto atteso; in caso contrario, si arresta notificando il fallimento del caso di test.

Inoltre, per simulare alcune situazioni di errore, sono stati prodotti pacchetti e comandi di aggiornamento specifici:

- **wumIncomplete**: versione modificata del comando wum che invia solo il 50% dei frammenti di un aggiornamento al WTP, lasciandolo in uno stato BUSY ad attendere gli altri frammenti;
- **capwap0.93.1.cup**: pacchetto che contiene una vecchia versione di OpenCAPWAP (in realtà sempre la 0.93.2, ma nel CUD è indicata la 0.93.1);
- **broken-MissingCUD.cup**: pacchetto malformato che non contiene il descrittore di aggiornamento *update.cud*;
- **broken-BadCUD.cup**: pacchetto che contiene un descrittore di aggiornamento, *update.cud*, illegale;
- **broken-BadPackage.cup**: pacchetto malformato (dati binari casuali, non è un archivio compresso con gzip);
- **broken-FailingPreScript.cup**: pacchetto che contiene uno script di pre-aggiornamento che fallisce sempre;
- **broken-FailingPostScript.cup**: pacchetto che contiene uno script di post-aggiornamento che fallisce sempre;
- **capwap-0.93.3.cup**: pacchetto che contiene una nuova versione di OpenCAPWAP ((in realtà sempre la 0.93.2, ma nel CUD è indicata la 0.93.3).

Il modello per la batteria di test usata, composta da 18 test individuali, è mostrato in Figura 27. I comandi vanno eseguiti nell'ordine in cui sono presentati. La batteria mira a stressare il sistema di aggiornamento e a verificare come esso si comporta in situazioni di errore. Ad esempio, i test 04-08 verificano cosa accade se viene inviato un pacchetto di aggiornamento non valido. I test 09-13 verificano invece la capacità del sistema di ripristinare una versione funzionante del WTP se l'aggiornamento fallisce. Ancora, i test 14-16 verificano se il sistema può essere riportato ad uno stato consistente qualora avvenissero problemi nel trasferimento del pacchetto di aggiornamento.

ID	Comando	Esito atteso
01	wum -w <id> -c version	0.93.2
02	wum -w <id> -c update -f capwap-0.93.1.cup	RICHIESTA RIFIUTATA
03	wum -w <id> -c version	0.93.2
04	wum -w <id> -c update -f broken-MissingCUD.cup	AGGIORNAMENTO NON ESEGUITO
05	wum -w <id> -c version	0.93.2
06	wum -w <id> -c update -f broken-BadCUD.cup	AGGIORNAMENTO NON ESEGUITO
07	wum -w <id> -c version	0.93.2
08	wum -w <id> -c update -f broken-BadPackage.cup	AGGIORNAMENTO NON ESEGUITO
09	wum -w <id> -c version	0.93.2
10	wum -w <id> -c update -f broken-FailingPreScript.cup	AGGIORNAMENTO NON ESEGUITO
11	wum -w <id> -c version	0.93.2
12	wum -w <id> -c update -f broken-FailingPostScript.cup	AGGIORNAMENTO NON ESEGUITO
13	wum -w <id> -c version	0.93.2
14	wumIncomplete -w <id> -c update -f capwap0.93.3.cup	WTP Update System resta nello stato BUSY
15	wum -w <id> -c update -f capwap0.93.3.cup	RICHIESTA RIFIUTATA (perché in stato BUSY)
16	wum -w <id> -c cancel	RICHIESTA ACCETTATA
17	wum -w <id> -c update -f capwap0.93.3.cup	AGGIORNAMENTO TRASFERITO CON SUCCESSO
18	wum -w <id> -c version	0.93.3

Figura 27: Test Cases per il sistema di aggiornamento dei WTP

La batteria è stata ripetuta su ciascun WTP dieci volte; non si sono manifestati particolari malfunzionamenti durante alcuna di queste ripetizioni. Ogni volta, in corrispondenza delle situazioni di errore simulate, il sistema è sempre stato in grado di tornare ad uno stato consistente e operativo (ad esempio, ripristinando una vecchia versione del WTP). Questa assenza di malfunzionamenti è dovuta al fatto che, durante lo sviluppo, sono stati eseguiti numerosi test di unità per verificare che le singole componenti del sistema che si stavano sviluppando svolgessero il proprio lavoro correttamente.

5.2 Conclusioni e sviluppi futuri

In questo lavoro ho ripercorso le fasi principali che hanno portato a due nuovi rilasci su *Sourceforge.org* di OpenCAPWAP, un'implementazione libera del protocollo CAPWAP per la gestione centralizzata di Access Point Wireless. Il primo di questi rilasci risolve numerosi bug presenti da tempo e legati, in particolare, alla gestione della memoria. Il secondo introduce una funzionalità per l'aggiornamento centralizzato del software di controllo del WTP.

La fase di analisi e correzione ha portato all'isolamento e alla catalogazione di venti bug; per ciascuno di essi sono state implementante delle *patch*, poi convogliate nella versione 0.92 di OpenCAPWAP. Due fattori sono stati molto importanti per il successo di questa fase: da un lato, la sintesi e l'impiego di una metodologia sistematica che offrisse delle linee guida per un'analisi sufficientemente rigorosa; dall'altro, la scelta e lo studio degli strumenti e delle tecnologie più adeguate per affrontare il debugging.

Nella fase di progettazione e implementazione del sistema di aggiornamento dei WTP si è cercato di applicare l'analogo informatico del rasoio di Ockham: il motto Unix *KISS – Keep It Simple, Stupid*. Questo ha portato a progettare un'architettura essenziale e funzionale che potesse però, al tempo stesso, essere sufficientemente scalabile da adattarsi bene ai possibili sviluppi futuri di OpenCAPWAP.

Al momento, il sistema di aggiornamento qui presentato, grazie al lavoro di un altro tesista, sta per essere integrato all'interno di un'*interfaccia web*; tale interfaccia sarà adottata presso il Consorzio *CASPUR* per amministrare in modo più semplice e diretto gli apparati della rete wireless *Provincia Wi-Fi.*

Appendice
Elenco dettagliato dei bug

In questa appendice è riportato l'elenco completo dei bug riscontrati e risolti, con allegati, dove possibile, i report di Valgrind.

ID	**Data**	**Componente**	**Stato**	**Patch**
UMR01	12/10/09	AC	Falso Positivo	--

Apparentemente, eseguendo l'AC sotto il controllo di valgrind, si verificano circa 150.000 accessi a zone di memoria non inizializzate. I report di valgrind sono di questo tipo:

```
==2793==  Conditional jump or move depends on uninitialised value(s)
==2793==    at 0x80ECD77: RSA_padding_add_PKCS1_type_2 (in
/home/skorpio/build/capwap-0.91/WTP)
```

Nell'analisi del problema, si è scoperto che questi messaggi di errore sono in realtà falsi positivi dovuti all'uso della libreria OpenSSL. Quando vengono chiamate le routine PRNG per generare numeri casuali, il contenuto di alcuni buffer viene mischiato nell'entropy pool: per questa ragione non importa se i buffer sono inizializzati o meno. (REF-> *http://openssl.org/support/faq.html#PROG14*)

ID	**Data**	**Componente**	**Stato**	**Patch**
TRL01	14/10/09	AC	Risolto	`capwap-TRL01-LE01.patch`

Dopo aver usato il comando di amministrazione remoteUci.py un numero consistente di volte (~300), non è più possibile adoperarlo: ogni volta che si prova a riutilizzarlo il sistema risponde col seguente messaggio di errore:

```
$ ./remoteUci.py -c wtps
Something Wrong Happened While Receiving Data From The AC Server
```

Il problema riguarda un l'esaurimento di risorse nella gestione dei thread. Per ogni richiesta eseguita col comando remoteUci.py, il thread di interfaccia dell'AC (ACInterface.c) genera un nuovo thread per servire tale richiesta. Dopo aver fatto ciò, se ne disinteressa completamente. Non viene eseguita ne' una join ne' il thread generato viene messo nello stato *detach*.

ID	Data	Componente	Stato	Patch
LE01	14/10/09	AC	Risolto	`capwap-TRL01-LE01.patch`

Questo errore logico è stato scoperto in fase di risoluzione del bug TRL01. Il thread di interfaccia verso il comando remoteUci.py gestisce un numero massimo di richieste contemporanee, superato il quale all'utente viene visualizzato il seguente messaggio:

```
$ ./remoteUCI.py -c wtps
The AC Server's Client Queue Is Currently Full
```

A causa di questo bug, il messaggio potrebbe essere visualizzato anche quando in realtà la coda delle richieste dell'AC non è piena.
Il problema è dovuto a una scorretta gestione da parte del thread di interfaccia di una situazione di errore. Quando arriva una richiesta, il contatore delle richieste che è ancora possibile servire (*numSocketFree*) viene decrementato e quindi si prova a creare il nuovo thread. Se la creazione del thread fallisce, il contatore non viene mai re-incrementato.

ID	Data	Componente	Stato	Patch
ML01	15/10/09	WTP	Risolto	`capwap-ML01.patch`

Durante gli stress test del sistema, dopo diverse ore di attività, è stato osservato che il WTP consuma una quantità sempre maggiore di memoria di sistema sempre maggiore (~30% dopo 20 ore di attività).

Valgrind report: (dopo ~1min di attività)

```
==3239== 41,052 (7,464 direct, 33,588 indirect) bytes in 933 blocks are definitely lost in loss record 39 of 42
==3239==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3239==    by 0x80550D0: CWParseVendorMessage (WTPRunState.c:1017)
==3239==    by 0x805536E: CWParseConfigurationUpdateRequest (WTPRunState.c:1119)
==3239==    by 0x8053B5D: CWWTPManageGenericRunMessage (WTPRunState.c:279)
==3239==    by 0x8053962: CWWTPEnterRun (WTPRunState.c:202)
==3239==    by 0x804ABC0: main (WTP.c:447)

==5632== 43,056 (14,352 direct, 28,704 indirect) bytes in 1,196 blocks are definitely lost in loss record 39 of 41
==5632==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==5632==    by 0x805E284: CWParseUCIPayload (CWVendorPayloadsWTP.c:46)
==5632==    by 0x805E3E2: CWParseVendorPayload (CWVendorPayloadsWTP.c:74)
==5632==    by 0x805518B: CWParseVendorMessage (WTPRunState.c:1047)
==5632==    by 0x8055396: CWParseConfigurationUpdateRequest (WTPRunState.c:1132)
==5632==    by 0x8053B64: CWWTPManageGenericRunMessage (WTPRunState.c:280)
==5632==    by 0x8053962: CWWTPEnterRun (WTPRunState.c:202)
==5632==    by 0x804ABC0: main (WTP.c:447)
```

Il problema riguarda un *memory leakage* che si verifica nella gestione delle richieste di *ConfigurationUpdate* di tipo *VendorSpecificPayload* implementate per gestire i *payload* UCI. Essenzialmente, vengono allocati dei buffer temporanei che vengono successivamente copiati nel messaggio da inviare dalla CWAssembleConfigurationUpdateResponse(). La patch risolve il problema liberano i buffer non appena il loro contenuto è stato copiato.

ID	Data	Componente	Stato	Patch
TRL02	15/10/09	CW	Risolto	`capwap-TRL02.patch`

Nel corso della risoluzione di alcuni bug, si è scoperto che il thread timer viene creato ma, non essendo mai fatta una join(), le sue risorse non vengono liberate fino al momento della terminazione(ref TRL01). Questo non costituisce un reale leakage di risorse, in quanto il timer dovrebbe vivere per tutta la durata dell'applicazione. Ad ogni modo, per mantenere il codice consistente, si è scelto di mettere il thread in stato *detach* al momento della creazione.

ID	Data	Componente	Stato	Patch
ML02	16/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2529== 0 bytes in 1 blocks are definitely lost in loss record 1 of 38
==2529==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==2529==    by 0x8052EEB: CWParseConfigureResponseMessage
(WTPConfigureState.c:251)
==2529==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==2529==    by 0x8052861: CWWTPEnterConfigure (WTPConfigureState.c:66)
==2529==    by 0x804ABEA: main (WTP.c:441)
```

La CWParseConfigureResponseMessage() alloca un array di strutture di tipo CWRadioOperationalInfoValues che non viene mai liberato. Il luogo corretto in cui rilasciare la memoria risulta essere la funzione WTPParseConfigureResponseMessage(), in cui vengono rilasciate anche altre zone di memoria facenti parte della stessa struttura.

ID	Data	Componente	Stato	Patch
ML03	16/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2677== 7 bytes in 1 blocks are definitely lost in loss record 3 of 37
==2677==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==2677==    by 0x805904E: CWProtocolRetrieveStr (CWProtocol.c:115)
==2677==    by 0x805AD9F: CWParseACName (CWProtocol.c:784)
==2677==    by 0x805241E: CWParseJoinResponseMessage (WTPJoinState.c:344)
==2677==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==2677==    by 0x8051DAF: CWWTPEnterJoin (WTPJoinState.c:176)
==2677==    by 0x804ABDF: main (WTP.c:438)
```

La funzione CWParseACName() legge il nome dell'AC a cui il WTP si sta associando nel campo name della variabile ACInfoPtr che a sua volta è un campo della struttura CWProtocolJoinResponseValues. Quest'ultima struttura è una variabile locale della funzione CWWTPEnterJoin() e viene usata come buffer per il *parsing* dei messaggi di Join che poi vengono copiati in una struttura globale. Usciti dalla CWWTPEnterJoin(), si perde ogni riferimento alla zona di memoria in cui è conservato il nome dell'AC.

ID	Data	Componente	Stato	Patch
ML04	16/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2859== 8 bytes in 1 blocks are definitely lost in loss record 4 of 36
==2859==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==2859==    by 0x806019D: CWBindingParseConfigureResponse (WTPBinding.c:527)
==2859==    by 0x80530A5: CWParseConfigureResponseMessage
(WTPConfigureState.c:293)
==2859==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==2859==    by 0x8052885: CWWTPEnterConfigure (WTPConfigureState.c:66)
==2859==    by 0x804ABEA: main (WTP.c:441)
```

La CWBidingParseConfigureResponse() alloca un buffer di tipo CWBindingConfigurationRequestValues e ne conserva un riferimento nella variabile locale *values* della funzione CWWTPEnterConfigure(). Al termine di questa funzione, viene perso il riferimento a questo oggetto.

ID	Data	Componente	Stato	Patch
ML05	16/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==3138== 8 bytes in 1 blocks are definitely lost in loss record 4 of 35
==3138==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3138==    by 0x8052F5B: CWParseConfigureResponseMessage
(WTPConfigureState.c:256)
==3138==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==3138==    by 0x8052885: CWWTPEnterConfigure (WTPConfigureState.c:66)
==3138==    by 0x804ABEA: main (WTP.c:441)
```

La CWParseConfigureResponseMessage() alloca un buffer di tipo WTPDecryptErrorReportValues e ne conserva un riferimento nella variabile locale values dela funzione CWWTPEnterConfigure(). Al termine di questa funzione, viene perso il riferimento a questo oggetto.

ID	Data	Componente	Stato	Patch
ML06	16/10/09	WTP	Risolto	`capwap-GENERAL.patch`

`Valgrind report`:

```
==3242== 10 bytes in 1 blocks are definitely lost in loss record 8 of 34
==3242==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3242==    by 0x8050806: CWWTPGetACNameWithIndex (WTPProtocol_User.c:345)
==3242==    by 0x804C901: CWAssembleMsgElemACNameWithIndex (WTPProtocol.c:64)
==3242==    by 0x8052A2D: CWAssembleConfigureRequest (WTPConfigureState.c:119)
==3242==    by 0x804A3F9: CWWTPSendAcknowledgedPacket (WTP.c:169)
==3242==    by 0x8052885: CWWTPEnterConfigure (WTPConfigureState.c:66)
==3242==    by 0x804ABEA: main (WTP.c:441)
```

```
==3242== 12 bytes in 1 blocks are definitely lost in loss record 10 of 34
==3242==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3242==    by 0x8050885: CWWTPGetACNameWithIndex (WTPProtocol_User.c:349)
==3242==    by 0x804C901: CWAssembleMsgElemACNameWithIndex (WTPProtocol.c:64)
==3242==    by 0x8052A2D: CWAssembleConfigureRequest (WTPConfigureState.c:119)
==3242==    by 0x804A3F9: CWWTPSendAcknowledgedPacket (WTP.c:169)
```

```
==3242==    by 0x8052885: CWWTPEnterConfigure (WTPConfigureState.c:66)
==3242==    by 0x804ABEA: main (WTP.c:441)
```

Nella CWWTPGetACNameWithIndex vengono allocate due stringhe con la macro CW_CREATE_STRING_FROM_STRING_ERR e vengono assegnate ai campi ACName dell'array ACNameIndex (array di strutture CWACNamesWithIndex). Successivamente, nella funzione CWAssembleMsgElemACNameWithIndex(), dopo aver assemblato il messaggio, viene liberato l'array ACNameIndex, ma non vengono liberati prima i campi ACName della struttura CWACNamesWithIndex in cui erano stati salvati i puntatori alle stringhe.

ID	Data	Componente	Stato	Patch
LE02	19/10/09	CW	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2663== 32 bytes in 3 blocks are definitely lost in loss record 12 of 32
==2663==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==2663==    by 0x8057024: CWParseTheFile (CWConfigFile.c:157)
==2663==    by 0x80572C7: CWParseConfigFile (CWConfigFile.c:201)
==2663==    by 0x804AC7B: CWWTPLoadConfiguration (WTP.c:491)
==2663==    by 0x804AA49: main (WTP.c:387)
```

Questo *memory leak* indicato da Valgrind è dovuto ad un errore logico nella procedura di *parsing* del file di configurazione. La CWParseConfigFile() invoca la CWParseTheFile() due volte: alla prima invocazione viene passato un flag che chiede alla funzione solo di contare gli ACAddress e i path; alla seconda invocazione non viene passato alcun flag e la funzione, in base al conteggio precedente, alloca i buffer ed esegue effettivamente il *parsing*.
A causa di un errore logico, però, i buffer che servono per contenere i valori di configurazione di tipo CW_STRING vengono allocati anche alla prima invocazione della funzione. Quindi, durante la successiva invocazione della CWParseTheFile(), questi buffer vengono riallocati sovrascrivendo i puntatori ai buffer precedentemente allocati e perdendo quindi ogni riferimento a quelle zone di memoria.

ID	Data	Componente	Stato	Patch
ML07	19/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==3662== 40 (32 direct, 8 indirect) bytes in 1 blocks are definitely lost in loss record 12 of 31
==3662==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3662==    by 0x804EF6A: CWParseACDescriptor (WTPProtocol.c:746)
==3662==    by 0x80516A6: CWParseDiscoveryResponseMessage
(WTPDiscoveryState.c:500)
==3662==    by 0x8050ED7: CWReceiveDiscoveryResponse (WTPDiscoveryState.c:266)
==3662==    by 0x8050D8B: CWReadResponses (WTPDiscoveryState.c:213)
==3662==    by 0x8050C3E: CWWTPEnterDiscovery (WTPDiscoveryState.c:158)
==3662==    by 0x804ABC9: main (WTP.c:432)
```

```
==4254== 40 bytes in 4 blocks are definitely lost in loss record 11 of 29
==4254==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==4254==    by 0x805919E: CWProtocolRetrieveRawBytes (CWProtocol.c:128)
==4254==    by 0x804F03C: CWParseACDescriptor (WTPProtocol.c:755)
```

```
==4254==    by 0x80516A6: CWParseDiscoveryResponseMessage
(WTPDiscoveryState.c:500)
==4254==    by 0x8050ED7: CWReceiveDiscoveryResponse (WTPDiscoveryState.c:266)
==4254==    by 0x8050D8B: CWReadResponses (WTPDiscoveryState.c:213)
==4254==    by 0x8050C3E: CWWTPEnterDiscovery (WTPDiscoveryState.c:158)
==4254==    by 0x804ABC9: main (WTP.c:432)
```

Durante lo stato di Discovery, la funzione CWParseACDescriptor alloca un array di strutture di tipo CWACVendorInfoValues collegato alla variabile globale gACInfoPtr. Successivamente, durante lo stato di Join, la funzione CWSaveJoinResponseMessage() sovrascrive il puntatore a questo array con l'indirizzo di un nuovo oggetto senza preoccuparsi di liberare la memoria precedentemente allocata.

ID	**Data**	**Componente**	**Stato**	**Patch**
ML08	19/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==4291== 20 bytes in 1 blocks are definitely lost in loss record 16 of 29
==4291==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==4291==    by 0x8052556: CWParseJoinResponseMessage (WTPJoinState.c:379)
==4291==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==4291==    by 0x8051DFB: CWWTPEnterJoin (WTPJoinState.c:176)
==4291==    by 0x804ABDF: main (WTP.c:438)
```

La CWParseJoinResponseMessage() alloca l'array Ipv4Addresses della struttura ACInfoPtr della risposta di Join; successivamente, usciti dalla CWWTPEnterJoin(), perdiamo ogni riferimento a questa zona di memoria (vedi ML03 per maggiori dettagli).

ID	**Data**	**Componente**	**Stato**	**Patch**
ML09	19/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==4312== 32 bytes in 2 blocks are definitely lost in loss record 16 of 28
==4312==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==4312==    by 0x8059222: CWProtocolRetrieveRawBytes (CWProtocol.c:128)
==4312==    by 0x804F3D2: CWParseACIPv6List (WTPProtocol.c:807)
==4312==    by 0x8052E91: CWParseConfigureResponseMessage
(WTPConfigureState.c:218)
==4312==    by 0x804A5E2: CWWTPSendAcknowledgedPacket (WTP.c:235)
==4312==    by 0x805297D: CWWTPEnterConfigure (WTPConfigureState.c:66)
==4312==    by 0x804ABEA: main (WTP.c:441)
```

La funzione CWProtocolRetriveRawBytes() restituisce un buffer allocato nell'*heap*; la funzione CWParseACIPv6List() gestisce in modo errato il valore di ritorno, copiandone il contenuto in un nuovo buffer e perdendo ogni riferimento al vecchio buffer:

```
CW_COPY_MEMORY(&((valPtr->ACIPv6List)[i]), CWProtocolRetrieveRawBytes(msgPtr, 16),
16);
```

ID	**Data**	**Componente**	**Stato**	**Patch**
UMR02	19/10/09	WTP	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2656== Thread 3:
==2656== Syscall param socketcall.bind(my_addr.) points to uninitialised byte(s)
==2656==    at 0x606A07: bind (in /lib/libc-2.10.1.so)
==2656==    by 0x6D0934: start_thread (in /lib/libpthread-2.10.1.so)
==2656==    by 0x60594D: clone (in /lib/libc-2.10.1.so)
==2656==  Address 0x5825b54 is on thread 3's stack
```

Il thread dedicato alla ricezione dei frame nel WTP non azzera la struttura *sockaddr* prima di invocare la funzione *bind*().

ID	Data	Componente	Stato	Patch
LE03	19/10/09	CW	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==2919== 32 bytes in 2 blocks are definitely lost in loss record 11 of 34
==2919==    at 0x4004E5C: calloc (vg_replace_malloc.c:397)
==2919==    by 0x805D199: get_ifi_info (CWStevens.c:270)
==2919==    by 0x8054526: CWNetworkInitSocketServerMultiHomed
(ACMultiHomedSocket.c:117)
==2919==    by 0x804A422: CWACInit (AC.c:158)
==2919==    by 0x804A273: main (AC.c:99)
```

Questo *memory leak* è causato da un errore logico nella get_ifi_info(). A causa della mancanza di un break in uno switch-case, della memoria allocata viene sovrascritta da un nuovo puntatore.

ID	Data	Componente	Stato	Patch
ML11	19/10/09	AC	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==3218== 8 bytes in 2 blocks are definitely lost in loss record 5 of 33
==3218==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3218==    by 0x80595EA: CWProtocolRetrieveRawBytes (CWProtocol.c:128)
==3218==    by 0x8057811: CWParseWTPBoardData (ACProtocol.c:550)
==3218==    by 0x804D222: CWParseDiscoveryRequestMessage (ACDiscoveryState.c:154)
==3218==    by 0x804B2EB: CWACManageIncomingPacket (ACMainLoop.c:165)
==3218==    by 0x805584F: CWNetworkUnsafeMultiHomed (ACMultiHomedSocket.c:525)
==3218==    by 0x804B17C: CWACEnterMainLoop (ACMainLoop.c:97)
==3218==    by 0x804A278: main (AC.c:100)
```

```
==3173== 40 (32 direct, 8 indirect) bytes in 1 blocks are definitely lost in loss record 12 of 34
==3173==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3173==    by 0x8057723: CWParseWTPBoardData (ACProtocol.c:544)
==3173==    by 0x804D222: CWParseDiscoveryRequestMessage (ACDiscoveryState.c:154)
==3173==    by 0x804B2EB: CWACManageIncomingPacket (ACMainLoop.c:165)
==3173==    by 0x805582B: CWNetworkUnsafeMultiHomed (ACMultiHomedSocket.c:525)
==3173==    by 0x804B17C: CWACEnterMainLoop (ACMainLoop.c:97)
==3173==    by 0x804A278: main (AC.c:100)
```

Nella CWParseBoardData vengono allocati dei buffer (in particolare l'array vendorInfos della struttura CWWTPVendorInfo); i puntatori a questa struttura sono mantenuti nella variabile locale values della funzione CWACManageIncomingPacket e devono essere

liberati dalla funzione CWDestroyDiscoveryRequestValues().

ID	Data	Componente	Stato	Patch
ML12	19/10/09	AC	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==3250== 236 (24 direct, 212 indirect) bytes in 2 blocks are definitely lost in loss record 23 of
32
==3250==    at 0x4006F3D: malloc (vg_replace_malloc.c:207)
==3250==    by 0x805A1B3: CWAssembleMessage (CWProtocol.c:414)
==3250==    by 0x804DB37: CWAssembleJoinResponse (ACJoinState.c:214)
==3250==    by 0x804D6E1: ACEnterJoin (ACJoinState.c:115)
==3250==    by 0x804C0FB: CWManageWTP (ACMainLoop.c:503)
==3250==    by 0x6D0934: start_thread (in /lib/libpthread-2.10.1.so)
==3250==    by 0x60594D: clone (in /lib/libc-2.10.1.so)
```

Il leak avviene qui, come in altri punti, perchè la CWAssembleMessage() alloca un array di messaggi nel campo messages della struttura CWWTPManager relativa al WTP che il thread sta gestendo (gWTPs(WTPIndex).messages). Il problema è che, da nessuna parte, ci si occupa di liberare questa memoria dopo che i messaggi sono stati inviati. Quindi, quando la CWAssembleMessage() verrà richiamata da altre parti del codice, il puntatore messages verrà semplicemente sovrascritto perdendo ogni riferimento alla memoria precedentemente allocata.

ID	Data	Componente	Stato	Patch
UMR03	20/10/09	AC	Risolto	`capwap-GENERAL.patch`

Valgrind report:

```
==3157== Syscall param rt_sigaction(act->sa_mask) points to uninitialised byte(s)
==3157==    at 0x6D94A7: __libc_sigaction (in /lib/libpthread-2.10.1.so)
==3157==    by 0x6D9572: sigaction (in /lib/libpthread-2.10.1.so)
==3157==    by 0x804B087: CWACEnterMainLoop (ACMainLoop.c:70)
==3157==    by 0x804A278: main (AC.c:100)
==3157==  Address 0xbee921f4 is on thread 1's stack
==3157==
==3157== Syscall param rt_sigaction(act->sa_mask) points to uninitialised byte(s)
==3157==    at 0x6D94A7: __libc_sigaction (in /lib/libpthread-2.10.1.so)
==3157==    by 0x6D9572: sigaction (in /lib/libpthread-2.10.1.so)
==3157==    by 0x804B0B6: CWACEnterMainLoop (ACMainLoop.c:75)
==3157==    by 0x804A278: main (AC.c:100)
==3157==  Address 0xbee921f4 is on thread 1's stack
```

La struttura *sigaction* non è stata inizializzata correttamente e quindi, all'atto dell'invocazione della system call, vengono passati al kernel valori casuali (il campo interessato è la maschera dei segnali da bloccare durante l'esecuzione del signal handler).

ID	Data	Componente	Stato	Patch
LE04	20/11/09	AC	Risolto	- -
Durante l'implementazione del sistema di aggiornamento discusso nel capitolo 4, sono stati scoperti dei bug nell'interfaccia dell'AC verso le applicazioni. Essenzialmente, non si controlla mai che le read() e le write() leggano o scrivano sui socket tutti i byte indicati. Questo può portare a diversi errori di difficile individuazione e correzione. Il problema è stato risolto utilizzando dei *wrapper*, Readn() e Writen(), che continuano, in un ciclo, a cercare di ricevere o spedire tutto il blocco dati riportando un errore in caso di fallimento.				

Bibliografia

[1] : James F. Kurose, Keith W. Ross, *Computer Networking: A Top-Down Approach*, 2007

[2] : M. Bernaschi, F. Cacace, G. Iannello, M. Vellucci, L. Vollero, *OpenCapwap: An Open Source Capwap Implementation forthe Management and Configuration of Wi-Fi Hot-Spots*, 2008

[3] : P. Calhoun, M. Montemurro, D. Stanley, *Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification*, 2009

[4] : S. Govindan, H. Cheng, ZH. Yao, WH. Zhou, L. Yang, *Objectives forControl and Provisioning of Wireless Access Points (CAPWAP)*, 2006

[5] : P. Calhoun, M. Montemurro e D. Stanley , *CAPWAP Protocol Binding for IEEE 802.11 (v. 7), Internet Draft*, 2008

[6] : Matteo Latini, *Disegno e Implementazione di un meccanismo per la gestione remota di reti wireless*, A.A. 2008/2009

[7] : http://www.technochakra.com/software-breakpoints/

[8] : Julian Seward, Nicholas Nethercote , *Valgrind: a framework for heavyweight dynamic binary instrumentation*, 2007

[9] : Steve Best, *Linux: Debugging and Performance Tuning*, 2005

[10] : http://en.wikipedia.org/wiki/Valgrind

[11] : Holger Keding, Markus Wloka, *The Developer's Guide to Debugging*, 2008

[12] : http://it.wikipedia.org/wiki/Debugging

Indice delle illustrazioni