

Parser Wireshark per Chronicle

Federica Di Giaimo, Filippo Lucchesi, Mariano Mancini

Abstract

L'analisi del traffico di rete è uno dei principali elementi di monitoraggio, essenziale per la *cyber*-sicurezza.

Il progetto si occupa della gestione e dell'analisi dei pacchetti catturati da *Wireshark*, per trasformare le informazioni ottenute in un formato che può essere analizzato dal servizio *cloud SIEM* di *Google*: *Google SecOps*.

Dopo aver analizzato tutti i formati rilevanti, sono stati testati più metodi di *parsing*. Per la scelta dei campi da inviare a *Google SecOps* sono stati valutati i possibili attacchi di cui si potesse mantenere una traccia durante lo *sniffing*, come l'intercettazione, la manipolazione dei dati e gli attacchi di negazione di servizio (attacchi di tipo *DoS*).

Nella stesura del *parser*, sono state convertite le informazioni generali e sui protocolli di rete:

- Indirizzi di sorgente e di destinazione: *IPv4*, *IPv6* e *MAC*.
- Informazioni sul protocollo di trasporto: *TCP* e *UDP*.
- Protocolli di rete: *ARP*, *ICMP*.
- Protocolli applicativi: *DNS*, *HTTP*, *TLS/SSL*.

L'intero processo di acquisizione ed analisi del traffico di rete è stato automatizzato con *Docker*, rendendo il flusso di lavoro robusto, portatile e facilmente replicabile.

Infine, vengono discussi possibili passi e ulteriori soluzioni per migliorare il *parser*, come l'aggiunta di campi che possano identificare nuovi attacchi, insieme a miglioramenti per estendere la compatibilità dell'applicazione.

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione | 4 |
| 2 | Gestione dei formati | 4 |
| 2.1 | Formati in <i>Wireshark</i> | 4 |
| 2.2 | Formati in <i>Google SecOps</i> | 4 |
| 3 | <i>Parsing</i> | 5 |
| 3.1 | Possibili soluzioni | 5 |
| 3.2 | Analisi delle minacce e attacchi potenziali | 5 |
| 3.2.1 | Attacchi di intercettazione e manipolazione di dati | 5 |
| 3.2.2 | Attacchi di <i>Denial of Service</i> | 6 |
| 3.2.3 | Vulnerabilità dei sistemi e iniezioni malevole | 6 |
| 3.3 | Conversione da <i>JSON</i> a <i>JSON-UDM</i> | 6 |
| 3.3.1 | Gestione indirizzi <i>IPv4</i> , <i>IPv6</i> e <i>MAC</i> | 7 |
| 3.3.2 | Gestione protocollo <i>ARP</i> | 7 |
| 3.3.3 | Gestione protocolli <i>TCP</i> e <i>UDP</i> | 7 |
| 3.3.4 | Gestione <i>DNS</i> e <i>MDNS</i> | 8 |
| 3.3.5 | Gestione protocollo <i>ICMP</i> | 9 |
| 3.3.6 | Gestione protocollo <i>TLS/SSL</i> | 9 |
| 3.3.7 | Gestione protocollo <i>HTTP</i> | 9 |
| 3.3.8 | Gestione dei <i>timestamp</i> | 10 |
| 3.4 | Soluzioni alternative | 10 |
| 3.4.1 | Scelta della libreria | 10 |
| 3.4.2 | <i>Parser PCAP</i> a <i>JSON-UDM</i> con <i>Pyshark</i> | 11 |
| 4 | Gestione dell'<i>output</i> | 12 |
| 4.1 | Funzione <code>write_to_multiple_files()</code> | 12 |
| 5 | Invio eventi | 13 |
| 5.1 | Autenticazione con <i>Google Cloud</i> | 13 |
| 5.2 | Invio eventi al <i>Cloud</i> | 13 |
| 6 | <i>Deployment con Docker</i> | 13 |
| 6.1 | Distribuzione tramite <i>Docker</i> | 14 |
| 6.2 | Funzionamento del file <i>entrypoint.sh</i> | 14 |
| 7 | Conclusione | 16 |
| 7.1 | Sviluppi futuri | 16 |
| 7.2 | Considerazioni finali | 16 |

1 Introduzione

La difesa e la sicurezza dei sistemi informatici si basa su processi metodici finalizzati al raggiungimento di risultati specifici mediante attività ben definite. *Google Security Operations* (*Google SecOps*), precedentemente noto come *Google Chronicle*, è uno strumento che analizza numerosi dati provenienti da *software* di analisi di pacchetti e permette alle organizzazioni di rilevare, analizzare e rispondere alle minacce [1]. Tuttavia, poiché *SecOps* non dispone di funzionalità native per la cattura del traffico di rete, è necessario ricorrere a strumenti specializzati come *Wireshark*. Quest'ultimo consente di acquisire in modo accurato i pacchetti, ma i dati raccolti sono espressi in formati complessi e non immediatamente compatibili con *SecOps*.

Gli obiettivi di questo progetto sono lo sviluppo di possibili soluzioni per la conversione di tali dati in un formato ideale per *Google SecOps*, in modo che siano immediatamente processati, e la valutazione dei campi più adatti da inviare per la rilevazione di possibili minacce.

2 Gestione dei formati

2.1 Formati in *Wireshark*

PCAP è il principale formato di *file* usato e supportato da *Wireshark* nella cattura dei pacchetti di rete [2], il cui salvataggio è possibile sia direttamente dalla *GUI*, che utilizzando *Wireshark* dal terminale. I *file* salvati in *PCAP* contengono tutte le informazioni dettagliate dei pacchetti ma, essendo scritti in binario, sono leggibili solo grazie a strumenti specializzati. Infatti, non sono adatti per essere inviati direttamente a *Google SecOps*, che accetta solo determinati tipi di *input* (2.2).

In *Wireshark* la cattura dei pacchetti di rete può essere salvata (o successivamente convertita) anche in formato *JSON*. Per la generazione dei *file* in tale formato, a differenza del formato *PCAP*, è necessario utilizzare *tshark*, la versione a linea di comando di *Wireshark*, inclusa già nel pacchetto d'installazione di quest'ultima [3].

2.2 Formati in *Google SecOps*

L'invio dei *log* a *Google SecOps* è reso possibile tramite la sua *Ingestion API* [4]. L'*API* accetta formati standardizzati secondo l'*Unified Data Model* (*UDM*) [5], *standard* che è possibile adottare sui *file .json*.

Un esempio di *file .json* accettato è il seguente: per ogni pacchetto catturato si catturano i campi **event**, contenenti le informazioni generali, e il campo **network** per i dettagli degli indirizzi e dei protocolli impiegati.

```
"event": {
  "type": "NETWORK_CONNECTION",
  "vendor_name": "Wireshark",
  "product_name": "Wireshark PacketCapture",
  "event_timestamp": "2024-12-20T16:04:03.199000+00:00"
},
"network": {
  "transport_protocol": "TCP",
  "ip": {
    "source": "192.168.1.116",
    "destination": "192.168.1.114",
    "ttl": "64"
  },
  "tcp": { "[...]" },
}
```

```
"frame": {
  "timestamp": "2024-12-20T16:04:03.199000+00:00",
  "length": "78",
  "protocols": "eth:ethertype:ip:tcp"
}
```

3 Parsing

3.1 Possibili soluzioni

Sono state analizzate più opzioni di conversione per fornire un formato leggibile a *Google SecOps*: una prima possibilità prevede l'utilizzo di un *file PCAP*, per il quale esistono varie librerie native in *Python* che consentono la conversione in formato *JSON-UDM*. Tuttavia, un'analisi approfondita dei pacchetti comporta l'utilizzo di molte risorse sia di computazione che temporali, appesantendo l'esecuzione dell'applicazione.

Un'alternativa consiste nell'utilizzare *tshark* per eseguire una prima elaborazione dei dati, che può includere la cattura del traffico di rete o la conversione di un *file PCAP* in *JSON*. Questo approccio riduce il carico di lavoro dello *script Python*, che non dovrà più occuparsi dell'intera conversione di formato, ma solo della riorganizzazione della struttura interna del *file* in *input* (nel modo specificato in 2.2).

3.2 Analisi delle minacce e attacchi potenziali

Definito il formato del *file* da utilizzare come *input* del *parser*, si è proceduto alla selezione dei campi più rilevanti. La valutazione delle informazioni da inviare è stata svolta considerando i principali attacchi a cui potrebbe essere soggetta una rete e, di conseguenza, individuando i campi che potrebbero contenerne delle tracce.

3.2.1 Attacchi di intercettazione e manipolazione di dati

Un attacco di tipo *Man-in-the-Middle (MitM)* può rappresentare una vera minaccia per la *privacy* di una rete. Questo tipo di attacco permette all'attaccante di inserirsi tra due sistemi in comunicazione, intercettando e potenzialmente alterando i messaggi scambiati [6] (proprio per questa caratteristica, prende il nome *Man-in-the-Middle*, ovvero "l'uomo nel mezzo"). È possibile rilevare anomalie nei pacchetti *TLS/SSL*, ad esempio controllando se si stanno utilizzando versioni obsolete (`tls.handshake.version`) o se si sta utilizzando `http` invece di `https`.

Un altro attacco informatico che riguarda la manipolazione di dati è lo *spoofing*, in cui l'attaccante falsifica le informazioni per apparire come un altro dispositivo [6]. Per controllare se si è soggetti a *IP spoofing* o *MAC spoofing* sono stati inseriti nel codice i campi *IP* e *MAC* di sorgente (rispettivamente `ip.src` e `eth.src`), nel caso *IP* non autorizzati cercassero di accedere a risorse interne. Anche in caso di *DNS spoofing* vi sono alcuni campi utili per individuare comportamenti anomali nel traffico dati. Il campo `dns.flags.response` permette a *Google SecOps* di controllare se la risposta *DNS* era sollecitata: in caso contrario, potrebbe trattarsi di un tentativo di *DNS spoofing*, in cui l'attaccante invia risposte *DNS* contraffatte per indirizzare il *client* verso un servizio malevolo, manipolando la risoluzione dei nomi di dominio. Un'altra tecnica sfruttata dagli attaccanti è utilizzare dei *Time-To-Live (TTL)* molto bassi nelle risposte *DNS*, in modo tale da avere un controllo dinamico della manipolazione *DNS* obbligando spesso il *client* a richiedere la risoluzione dei nomi del dominio. Buona prassi è inserire anche il campo `dns.qry.name`, per permettere a *Google SecOps* di filtrare i domini ed esaminare quelli più soggetti a *spoofing* [7].

3.2.2 Attacchi di *Denial of Service*

Tra i principali attacchi che sono identificati da un *IDS* vi sono il *Denial of Service (DOS)* e il *Distributed Denial of Service (DDoS)*. L'obiettivo di questi attacchi è di rendere una risorsa non disponibile, solitamente saturando la rete con un numero ingente di pacchetti [6]. Questo tipo di attacco può essere individuato da *Google SecOps* utilizzando i campi `ip.src` e `ip.dst` per controllare se un singolo *IP* (o un numero limitato di *IP*) ha inviato un numero elevato di richieste, e i campi `tcp.srcport` e `tcp.dstport` per monitorarne l'arrivo. Anche il campo `tcp.flags` può essere importante per identificare un attacco *DoS*, individuando specifici schemi di attacco (ad esempio un *SYN flood*). Infine, per il calcolo della frequenza delle richieste, viene inviato il campo `event_timestamp`, contenente il *timestamp* di ogni pacchetto.

3.2.3 Vulnerabilità dei sistemi e iniezioni malevole

Alcuni degli attacchi considerati di seguito sono presenti nella *top 10* dei rischi delle *Web Application* stilata da *OWASP* [8], un elenco riconosciuto a livello mondiale riguardante le minacce più pericolose, aggiornata l'ultima volta nel 2021. Al terzo posto vi sono gli attacchi di tipo *Injection*, ovvero i tentativi da parte dell'attaccante di inviare dati ad un'applicazione in modo tale da cambiare il significato dei comandi [6]. La più comune, l'*SQL Injection*, potrebbe essere individuata nell'*URL* della *HTTP Requests*, controllando se vi sono caratteri speciali come `'`, `.*` o `#`, presenti spesso in attacchi di manipolazione del codice. Possibili *injection* possono essere intercettati anche analizzando il campo `http.file_data`, nel caso l'attaccante avesse caricato una *SQL injection* all'interno di essa.

3.3 Conversione da *JSON* a *JSON-UDM*

Nei paragrafi sottostanti sono riportati delle sezioni di codice per convertire il *file* fornito da *tshark* e standardizzarlo in *UDM*. Nel caso in cui i campi che si tentano di convertire non sono presenti in *input* o non sono validi, il codice provvede a rimuoverli dal dizionario dell'*output*, garantendo maggiore leggibilità e alleggerendo il lavoro di *Google SecOps*.

Per l'esecuzione dello *script* è necessario specificare:

- Il *path* del *file .json* da convertire.
- Il *path* e il nome che si preferisce dare al *file* di *output*.

```
1 python3 json2udm.py <\path\to\input_file.json> <\path\to\output_file_name>
```

Nell'*output* sono stati inseriti alcuni parametri predefiniti per ogni evento, per indicare che sono stati forniti da *Wireshark*. È specificato anche il *timestamp*, la cui conversione è gestita dalla funzione `convert_timestamp`, descritta nella sezione apposita (3.3.8).

```
1 "event": {
2     "type": "NETWORK_CONNECTION",
3     "vendor_name": "Wireshark",
4     "product_name": "Wireshark PacketCapture",
5     **({"event_timestamp": convert_timestamp(frame.get("frame.time_utc"))})
6     if frame.get("frame.time_utc") else {}},
7 }
```

N.B.: Il codice descritto nelle prossime sezioni è stato ridotto per agevolarne la comprensione e la leggibilità. Sono stati rimossi blocchi `try...catch`, `print` e `logging`.

3.3.1 Gestione indirizzi *IPv4*, *IPv6* e *MAC*

Come spiegato nel paragrafo precedente (vedi 3.2), filtrare i campi di sorgente e di destinazione di ogni pacchetto è fondamentale per rilevare anomalie nel flusso dei pacchetti in rete. Nel *parser* sono stati gestiti gli indirizzi *MAC* e gli indirizzi *IP*, sia in *IPv4* che in *IPv6*.

```
1 #IPv4 address
2 **({"ip": {
3     **({"source": ip.get("ip.src")} if ip.get("ip.src") else {}),
4     **({"destination": ip.get("ip.dst")} if ip.get("ip.dst") else {}),
5     **({"ttl": ip.get("ip.ttl")} if ip.get("ip.ttl") else {}),
6 }} if ip else {}),
7
8 #IPv6 address
9 **({"ipv6": {
10     **({"source": ipv6.get("ipv6.src")} if ipv6.get("ipv6.src") else {}),
11     **({"destination": ipv6.get("ipv6.dst")} if ipv6.get("ipv6.dst") else {}),
12 }} if ipv6 else {}),
13
14 #MAC address
15 **({"eth": {
16     **({"source_mac": eth.get("eth.src")}
17     if eth.get("eth.src") else {}),
18     **({"destination_mac": eth.get("eth.dst")}
19     if eth.get("eth.dst") else {}),
20 }} if eth else {})
```

3.3.2 Gestione protocollo *ARP*

Sono stati presi in considerazione anche i protocolli *ARP*, i cui campi possono segnalare casi di *ARP spoofing* in caso di associazioni anomale tra indirizzi *MAC* e *IP*.

```
1 **({"arp": {
2     **({"source_mac": arp.get("arp.src.hw_mac")}
3     if arp.get("arp.src.hw_mac") else {}),
4     **({"source_ipv4": arp.get("arp.src.proto_ipv4")}
5     if arp.get("arp.src.proto_ipv4") else {}),
6     **({"destination_mac": arp.get("arp.dst.hw_mac")}
7     if arp.get("arp.dst.hw_mac") else {}),
8     **({"destination_ipv4": arp.get("arp.dst.proto_ipv4")}
9     if arp.get("arp.dst.proto_ipv4") else {}),
10 }} if arp else {})
```

3.3.3 Gestione protocolli *TCP* e *UDP*

A livello di trasporto dei pacchetti sono stati presi in considerazione sia il protocollo *TCP* che il protocollo *UDP*. Sono convertite le informazioni riguardo le porte della sorgente e della destinazione e, nel *TCP*, si riportano le *flag*, che hanno lo scopo di indicare lo stato della connessione *TCP*. Ciò può essere importante per indicare attività insolite, come l'invio ripetuto di pacchetti *TCP* con *flag* *SYN* (potrebbe trattarsi di un attacco *SYN Flood*) o *RST*.

```
1 #UDP protocol
2 *({"udp": {
3     **({"source_port": udp.get("udp.srcport")}
4     if udp.get("udp.srcport") else {}),
```

```

5     **({"destination_port": udp.get("udp.dstport")})
6         if udp.get("udp.dstport") else {}),
7 }} if udp else {}),
8
9 #TCP protocol
10 **({"tcp": {
11     **({"source_port": tcp.get("tcp.srcport")})
12         if tcp.get("tcp.srcport") else {}),
13     **({"destination_port": tcp.get("tcp.dstport")})
14         if tcp.get("tcp.dstport") else {}),
15     **({"flags": tcp.get("tcp.flags")})
16         if tcp.get("tcp.flags") else {}),
17 }} if tcp else {}))

```

3.3.4 Gestione DNS e MDNS

L'accesso di alcuni campi *DNS* è meno diretto: è necessario iterare tra i sottocampi di alcune chiavi. Il codice svolge questo compito con la funzione `print_dns`:

```

1 def print_dns(items,key):
2     results = []
3     for k, v in items:
4         if isinstance(v, dict):
5             result = v.get(key)
6             if result is not None:
7                 results.append(result)
8     return results if results else None

```

I campi considerati in questo caso sono essenziali per individuare attacchi come *DNS Spoofing*, *DDoS* o anche *DNS Tunneling*, un tipo di attacco che es filtra dati attraverso le richieste e le risposte *DNS*. Campi come `dns.qry.type` sono usati per individuare *record* e *payload* anomali all'interno del traffico in rete.

N.B.: Vista la poca leggibilità del seguente frammento di codice, si è preferito riportare solo la conversione di due campi. Per ottenere `dns.qry.type` si è iterato nel campo *Queries*, mentre per ottenere il campo `dns.flag.response` si è iterato in `dns.flags_tree`, sempre con la funzione `print_dns`.

```

1 **({"dns": {
2     **({"query": {
3         **({"name": print_dns(dns["Queries"].items(), "dns.qry.name")})
4             if "Queries" in dns and print_dns(dns["Queries"].items(),
5                 "dns.qry.name") is not None else {}),
6
7     #         convert "dns.qry.type" [...]
8
9     **({"ttl": print_dns(dns["Answers"].items(), "dns.resp.ttl")})
10        if "Answers" in dns and print_dns(dns["Answers"].items(),
11            "dns.resp.ttl") is not None else {}),
12
13    #         convert "dns.flag.response" [...]
14    }} if any(key in dns for key in ["Queries", "Answers", "dns.flags_tree"])
15    else {}),
16 }} if dns else {}))

```

La gestione dei campi *MDNS* è del tutto analoga a quella *DNS* (per questo non riportato del documento), con la differenza che non necessita del campo `dns.flags.response`. Infatti,

mentre *DNS* è strutturato su un modello *client-server*, quindi necessita di accedere ad un campo che indichi se il pacchetto inviato è una richiesta o una risposta, *MDNS* non ha una struttura centralizzata che gestisce le risposte, perciò non è essenziale distinguere i due tipi di pacchetti.

3.3.5 Gestione protocollo *ICMP*

È stato incluso anche il protocollo *ICMP* convertendo i campi `icmp.type` e `icmp.code`. Con tali campi è possibile rilevare un possibile *Ping Flood*. Permettono anche di filtrare e monitorare i messaggi di tipo *Destination Unreachable*, che potrebbero indicare tentativi di scansione da parte di un attaccante o problemi di rete [6].

```
1  **({"icmp": {
2      **({"type": icmp.get("icmp.type")} if icmp.get("icmp.type") else {}),
3      **({"code": icmp.get("icmp.code")} if icmp.get("icmp.code") else {}),
4  }} if icmp else {})
```

3.3.6 Gestione protocollo *TLS/SSL*

Per quanto riguarda il protocollo *TLS/SSL*, il codice presenta una versione base: sono stati convertiti i campi `tls.handshake.version` e `tls.record.version`, i quali specificano le versioni *TLS* negoziate. *Google SecOps* potrebbe rilevare situazioni di *Downgrade Attack*, nel caso sia stata forzata una versione più vecchia di *TLS*, come 1.0 o 1.1 invece che 1.2 o 1.3. Come per i campi *DNS* (vedi 3.3.4), anche per l'accesso a `tls.handshake.version` è stata definita una funzione *ad hoc*, `print_handshake`, che itera tra gli elementi di `tls.handshake` per accedere a quello richiesto.

Sviluppi futuri del codice potrebbero riguardare un'estensione delle informazioni convertite del campo *TLS*, come `tls.handshake.cipher_suite`, per individuare in modo più approfondito l'utilizzo di protocolli deboli, oppure i campi `tls.handshake.server_name` e `tls.cert.subject` che, se non coincidenti, potrebbero aiutare ad individuare un attacco *spoofing* in cui l'attaccante sta presentando un certificato non autentico.

```
1  "tls": {
2      **({"record_version": tls["tls.record"].get("tls.record.version")}
3          if "tls.record" in tls and tls["tls.record"].get("tls.record.version")
4          is not None else {}),
5
6      **({"handshake": {
7          **({"version": print_handshake(tls["tls.record"], "tls.handshake.version")}
8              if "tls.record" in tls and print_handshake(tls["tls.record"],
9                  ↪ "tls.handshake.version") is not None else {}),
9          }} if "tls.handshake" in tls["tls.record"] else {}),
10 } if tls else {},
```

3.3.7 Gestione protocollo *HTTP*

Anche per la gestione del protocollo *HTTP* è stata avviata una struttura base. Essa ottiene i *payload* del corpo dei messaggi inviati, attraverso il campo `http.file_data`, utile per l'individuazione di eventuali *Cross-Site Scripting (XSS)*. Convertendo anche l'`http.host` è possibile verificare anche se tali campi sono utilizzati per scrivere *query* non validate, catturando una possibile *SQL Injection*.

Un possibile miglioramento potrebbe consistere nel catturare anche le informazioni contenute nell'*URL* e il metodo della richiesta, in modo tale da intercettare possibili attacchi di *phishing* e individuare nell'*header* della richiesta ulteriori *SQL Injection*.

```

1  **({"http": {
2      **({"host": http.get("http.host")})
3      if http.get("http.host") else {}},
4      **({"file_data": http.get("http.file_data")})
5      if http.get("file_data") else {}},
6  }) if http else {}),

```

3.3.8 Gestione dei *timestamp*

I *timestamp* presenti nei *file JSON* di *Wireshark* devono essere convertiti nel formato RFC 3339 [9], una specifica di ISO 8601, uno *standard* internazionale per la rappresentazione di date e ore [10]. Tale conversione è eseguita dalla seguente funzione *convert_timestamp()*:

```

1  def convert_timestamp(timestamp_str):
2      dt = datetime.strptime(timestamp_str[:25], "%b %d, %Y %H:%M:%S.%f")
3      dt = dt.replace(tzinfo=timezone.utc)
4      iso_timestamp = dt.isoformat()

```

3.4 Soluzioni alternative

Come detto precedentemente, la soluzione più diretta consiste nella realizzazione di una conversione completa, sfruttando le varie librerie disponibili per la lettura e analisi dei *file PCAP*. Dopo un'attenta valutazione, si è scelto di adottare una riformattazione partendo dal *JSON*, in modo da alleggerire l'esecuzione e migliorarne le *performance*. Tuttavia, per giungere a questa decisione, sono state testate alcune tra le librerie per la conversione *PCAP* più utilizzate, sviluppando versioni base per effettuare una valutazione delle *performance*.

3.4.1 Scelta della libreria

Per costruire un *parser* con input i *file PCAP* in *Python* esistono diverse librerie. In questo caso si è deciso di considerare le tre principali per comprenderne vantaggi e svantaggi: *Pyshark*, *Scapy* e *Dpkt*.

Pyshark è una libreria *Python* basata su *tshark* che permette di catturare e analizzare pacchetti di rete sia da *file PCAP* che in tempo reale. Supporta diversi protocolli, consente di applicare filtri per selezionare pacchetti specifici ed è compatibile con varie versioni di *tshark* [11].

Scapy è più potente e flessibile, permettendo non solo l'analisi, ma anche la creazione e manipolazione dei pacchetti, ideale per simulazioni di rete avanzate [12].

Dpkt, scritta in *C*, è la più veloce ed efficiente, adatta all'elaborazione di grandi volumi di traffico, ma con meno funzionalità avanzate [13].

Sia *Scapy* che *Dpkt* sono più semplici nelle loro capacità di *parsing* e manipolazione dei pacchetti. Questa caratteristica, però, porta con sé un possibile svantaggio: potrebbero non essere in grado di gestire alcune varianti di *PCAP* più complesse o non *standard*, che potrebbero essere state create da versioni più recenti di *Wireshark* o con particolari configurazioni di cattura. *Pyshark*, come già accennato, si basa su *Tshark* ed è più robusto e tollerante anche in caso di pacchetti danneggiati.

Un confronto tra le tre librerie in termini di facilità d'uso, *performance* e funzionalità avanzate permette di evidenziarne le differenze principali:

| Qualità | Pyshark | Scapy | Dpkt |
|-------------------------------|---|---|--|
| Facilità d'uso | Molto semplice, sintassi intuitiva | Più complessa, ma molto flessibile | Semplice, ma meno versatile |
| Velocità e performance | Più lento (wrapper per tshark) | Prestazioni inferiori rispetto a Dpkt | Molto veloce, ideale per grandi <i>file PCAP</i> |
| Funzionalità avanzate | Limitato all'analisi e <i>parsing</i> dei pacchetti | Creazione, manipolazione e invio di pacchetti | Solo analisi, nessuna creazione di pacchetti |

La scelta della libreria più adatta dipende dal caso d'uso specifico. **Pyshark** è ideale quando si ha bisogno di una soluzione semplice per l'analisi dei *file PCAP* senza necessità di manipolare pacchetti. **Scapy** è la scelta migliore nel caso in cui si desidera maggiore controllo e flessibilità, ideale per *pentesting*, *fuzzing* e *testing* dei protocolli di rete. **Dpkt**, infine, è la scelta giusta quando le prestazioni e l'efficienza nell'elaborazione di grandi *file PCAP* sono una priorità.

3.4.2 Parser PCAP a JSON-UDM con Pyshark

Di seguito sono presentati alcuni frammenti di codice del *parser* che adopera la libreria **Pyshark**.

```
1 cap = pyshark.FileCapture(pcap_file)
```

L'oggetto di tipo `FileCapture` carica il *file PCAP* passato in *input* (`pcap_file`), permettendone in questo modo l'accesso e l'iterazione dei pacchetti presenti, rappresentati come oggetti **Packet**.

```
1 for packet in cap:
2     packet_info = {
3         "event": {
4             "type": "network_traffic",
5             [...]
6         },
7         "network": {
8             "protocol": None,
9             [...]
10        }
```

Nel blocco di codice succitato si crea il dizionario `packet_info`, in cui viene definita la struttura del *file* di *output* finale, come presentata in 2.2.

```
1 if "ETH" in packet:
2     eth_layer = packet.eth
3     packet_info["source"]["mac"] = getattr(eth_layer, "src", None)
4     [...]
5     extracted_data.append(packet_info)
```

Per ogni pacchetto i campi richiesti sono convertiti in oggetti *Python* da **Pyshark** e sono inseriti nel dizionario adeguatamente. Quando l'estrazione è completata il dizionario è aggiunto alla lista `extracted_data`.

```
1 json.dump(data, json_file, indent=4, default=str)
```

Infine la funzione `json.dump()` struttura i dati convertiti in modo tale da essere riportati su un *file* direttamente processabile da *Google SecOps*.

4 Gestione dell'*output*

Una volta terminata la conversione degli eventi presenti nel *file* di *input* e successivamente accodati in una lista denominata `udm_events`, si generano i *file* di *output*. Tuttavia, la scrittura dell'*output* non è immediata: ogni richiesta *HTTP* è accettata dall'*API* solo se l'*output* ha una dimensione inferiore a 1 MB [4]. Ad occuparsene è la funzione `write_to_multiple_files()`, a cui è passata la lista `udm_events` e il nome che si desidera dare al *file*, passato da subito sulla linea di comando (3.3).

4.1 Funzione `write_to_multiple_files()`

La seguente funzione si occupa di ovviare a questi limiti, andando a creare più di un *file* ogni volta che viene superato il limite di 1 MB. I *file* presentano la seguente nomenclatura:

- Il primo *file* è denominato `<output_file>_1`, anche nel caso in cui sia l'unico ad essere generato.
- I *file* successivi saranno denominati ordinatamente incrementando l'indice (`<output_file>_2`, `<output_file>_3`,...).

Il contatore è rappresentato dalla variabile `current_file_index`, inizializzato ad 1. La funzione itera nella lista di eventi, calcolandone la dimensione in *byte*.

```
1 def write_to_multiple_files(udm_events, base_output_file):
2     max_size_bytes = MAX_FILE_SIZE_MB * 1024 * 1024 # Convert MB to bytes
3
4     for event in (udm_events):
5         event_json = json.dump(event, indent=4)
6         event_size = len(event_json.encode("utf-8")) # Size in bytes
7     # [...]
```

Si controlla se l'evento può essere aggiunto nel *file* che si sta attualmente considerando. In caso contrario, il codice crea il *file* di *output* senza l'ultimo evento considerato, incrementa il contatore e inizializza le variabili per prepararsi alla generazione del nuovo *file*.

```
1 # Check if adding this event exceeds the size limit
2 if current_size + event_size >= max_size_bytes:
3     output_file = f"{base_output_file}_{current_file_index}.json"
4     with open(output_file, "w") as f:
5         json.dump(current_events, f, indent=4)
6     # Start a new file [...]
```

Se invece l'evento può essere aggiunto, è inserito nella lista `current_events`:

```
1 # Add the event to the current file's list
2 current_events.append(event)
3 current_size += event_size
```

Terminata l'iterazione, si genera un ultimo *file* nel caso vi siano eventi rimanenti.

```
1 # Write the last file if there are remaining events
2 if current_events:
3     output_file = f"{base_output_file}_{current_file_index}.json"
4     with open(output_file, "w") as f:
5         json.dump(current_events, f, indent=4)
```

5 Invio eventi

5.1 Autenticazione con *Google Cloud*

Google-auth è la libreria per l'autenticazione e la comunicazione con le *API* di *Google* per *Python* basata sul protocollo *OAuth 2.0* [14]. Per l'autenticazione di richieste al *client API* è necessario creare un *account* di servizio *Google*, generando una chiave *json* contenente le credenziali necessarie.

```
1 # Create credential object to authenticate API requests
2 credentials = service_account.Credentials.from_service_account_file(
3     ing_service_account_file,
4     scopes='https://www.googleapis.com/auth/malachite-ingestion')
```

Per garantire maggiore flessibilità, si è preferito dichiarare le credenziali in un altro *file*, *chronicle-api.conf*, presente nella stessa cartella del codice.

```
1 ING_SERVICE_ACCOUNT_FILE=/path/to/json/apikeys-demo.json
2 CUSTOMER_ID=01234567-89ab-cdef-0123-456789abcdef
```

5.2 Invio eventi al *Cloud*

Per l'invio di una richiesta all'*API* si specifica anzitutto l'*endpoint* regionale con cui comunicare:

```
1 # Regional endpoint for API call - Turin
2 INGESTION_API = "https://europe-west12-malachiteingestion-pa.googleapis.com"
```

Successivamente, si crea una sessione *HTTP* autorizzata, che include già i *token* d'accesso e l'*header Authorization*.

```
1 # Build an authorized HTTP session
2 http_session = requests.AuthorizedSession(credentials)
3
4 # Complete endpoint
5 url = f"{INGESTION_API}/v2/udmevents:batchCreate"
6
7 # Request body
8 body = {
9     "customerId": customer_id,
10    "events": json.loads(json_events),
11 }
12 response = http_session.request("POST", url, json=body)
```

6 Deployment con *Docker*

L'intero processo di acquisizione e elaborazione dei dati è automatizzato dal *container Docker*, rendendo il flusso di lavoro robusto, portatile e facilmente replicabile. In questo modo l'applicazione diventa più gestibile e configurabile in vari ambienti, riducendo al minimo le problematiche di compatibilità.

6.1 Distribuzione tramite *Docker*

Nel progetto, *Docker* è stato sfruttato per automatizzare l'avvio di *tshark* e gestirne la raccolta dei dati, poi effettuarne il *parsing* tramite lo *script Python*. Ne risulta un'automatizzazione completa, dalla raccolta all'analisi dei dati, rendendo la catena di operazioni eseguibile senza alcuno sforzo dell'utente e ripetibile in qualsiasi ambiente.

- **Dockerfile**: definisce l'immagine [15] del *container*, specificando i passaggi necessari per la sua configurazione. Questo include l'installazione di *tshark* e delle librerie necessarie oltre alla preparazione del sistema operativo virtualizzato.
- **Docker-Compose**: un *file YAML* [16] per semplificare la gestione dei volumi [17] e per dichiarare le risorse necessarie. In questo *file* sono specificate anche le variabili d'ambiente [18] per configurare il comportamento del *container*.
- **Script *entrypoint.sh***: il cuore del processo. Gestisce l'intero flusso di lavoro, dall'acquisizione dei pacchetti di rete alla loro elaborazione e successivo invio a *Google SecOps*.

6.2 Funzionamento del *file entrypoint.sh*

Lo *script* permette di automatizzare completamente il flusso di lavoro senza il rischio di incorrere in errori da parte degli operatori. Utilizzando *docker*, il progetto sarà altamente replicabile e manutenibile, evitando problemi molto comuni correlati all'adattamento a diversi ambienti.

```
1 FROM python:3.9-slim
2 RUN apt-get update && apt-get install -y tshark inotify-tools && \
3     apt-get clean && rm -rf /var/lib/apt/lists/*
4 WORKDIR /app
5 COPY json2udm.py /app/json2udm.py
6 #COPY ingestion_comm.py /app/ingestion_comm.py
7 #COPY chronicle-api.conf /app/chronicle-api.conf
8 COPY entrypoint.sh /app/entrypoint.sh
9 RUN chmod +x /app/entrypoint.sh
10 ENTRYPOINT ["/app/entrypoint.sh"]
```

All'avvio, il *container* eseguirà *entrypoint.sh*, che si occuperà di tutte le operazioni necessarie per estrapolare le informazioni utili dall'interfaccia di rete in uso, e ne effettuerà il *parsing* in modo da renderle compatibili con *Google SecOps*.

La prima fase del processo riguarda il controllo dei percorsi: lo *script* verifica se i volumi dichiarati corrispondono effettivamente all'ambiente virtualizzato, se le cartelle esistono e se contengono *file* che devono ancora essere elaborati. Se necessario, provvede alla loro traduzione prima di iniziare l'esecuzione di *tshark*.

Il prossimo passo consiste nella selezione automatica di un'interfaccia di rete valida, esaminando le disponibili in */sys/class/net/*. Se nessuna si trova nello stato di *up*, il *container* attende, riprovando ogni 5 secondi. Subito dopo, *tshark* entra in azione per catturare i pacchetti sulla rete utilizzando l'interfaccia specificata. I parametri di esecuzione (come la durata e i filtri) sono configurabili tramite variabili d'ambiente nel *file compose.yml*, consentendo così di adattare il comportamento dell'applicazione alle necessità specifiche del contesto.

La fase successiva riguarda il monitoraggio e l'elaborazione dei *file*: lo *script* si avvale di *inotifywait* [19] per tenere sotto controllo la cartella *./sniff* e rilevare i *file pcap* appena completati, monitorando l'occorrenza degli eventi *write_close*.

```
1 inotifywait -m -e close_write --format "%w%f" "$INPUT_DIR" | while read -r NEW_FILE;
   ↪ do
2     echo "File completed: $NEW_FILE"
```

```

3      # Small delay to ensure file is fully written
4      sleep 1
5      process_file "$NEW_FILE"
6  done

```

Una volta pronto, il file viene convertito in formato *JSON* mediante la funzione *bash* `process_files()`, che opera con *tshark*, per poi essere ulteriormente elaborato dallo *script* del *parser*, `json2udm.py`. In questo modo è reso compatibile con il formato *RFC 3339* richiesto da *Google SecOps*.

```

1      # Step 1: pcap -> json (move the file to trash if successful)
2      if tshark -r "$1" -T json > "$MID_DIR/$FILE"; then
3          mv "$1" "$TRASH_DIR/"
4      else [...] fi
5
6      # Step 2: json -> UDM (remove input file if successful)
7      if python3 /app/json2udm.py "$MID_DIR/$FILE" "$OUTPUT_DIR/$FILE"; then
8          echo "Processing successful, removing file: $MID_DIR/$FILE"
9          rm "$MID_DIR/$FILE"
10     else [...] fi

```

La terza fase consiste nell'invio dei risultati prodotti a *Google SecOps*. Se la seconda ha esito positivo, si ha l'esecuzione del secondo *script Python*, `ingestion_comm.py`, che effettua l'invio a *Google SecOps*:

```

1      # Step 3 (optional): sending results to Google Chronicle
2      if python3 /app/ingestion_comm.py "$OUTPUT_DIR/$FILE"; then
3          echo "Results successfully sent to Google Chronicle" [...] fi

```

Il *container* è configurato in tre volumi: uno per i *file* di *input* (contenenti i dati grezzi raccolti da *tshark*), uno come "cestino" (dove vengono spostati i *file pcap* processati, anziché eliminarli, permettendo ulteriori analisi) e uno per i *file* di *output* (i risultati elaborati).

Un quarto volume, utile principalmente per il *debug* e nascosto durante il normale funzionamento, è dedicato ai *file* temporanei, dove vengono conservati i *file JSON* intermedi. Questa separazione ottimizza la gestione dei dati e facilita l'impiego del *container* in scenari differenti.

```

1  wireshark2chronicle:
2      # [...]
3      volumes:
4          - ./sniff:/app/input
5          - ./processed:/app/trash
6          - ./chronicle:/app/output
7      environment:
8          - ROTATE=-b filesize:1024
9          - LIMITS=-c 20000

```

In caso di errore durante la conversione o l'elaborazione, il sistema segnala il problema e mantiene i *file* originali per una possibile risoluzione. Inoltre, lo *script* gestisce l'*output* dei *log*, reindirizzando tutte le informazioni rilevanti, così che possano essere consultate facilmente per il *debug* o l'analisi del comportamento del sistema.

7 Conclusione

Per affrontare il problema del *parsing*, erano disponibili principalmente due approcci: un *parsing* diretto, sfruttando le librerie *Python* disponibili per leggere e analizzare i file *PCAP*, oppure un metodo suddiviso in due fasi: una prima conversione in *JSON* tramite *tshark*, seguita da una seconda fase "adattamento" nel formato *JSON-UDM* richiesto da *Google SecOps*. Dopo un'attenta valutazione, è stata scelta la seconda strategia, in quanto ha permesso di ridurre il carico computazionale del *parsing*, accelerandone l'esecuzione.

Tuttavia, l'impiego di due strumenti distinti ha introdotto una maggiore complessità nella gestione delle dipendenze. Infatti, mentre nel primo approccio sarebbe stato sufficiente collegare direttamente le librerie necessarie, in questo caso è stata necessaria l'installazione di un ulteriore strumento dedicato. Per ovviare a questo problema, si è optato per una distribuzione tramite *container*, offrendo così un prodotto in cui tutte le dipendenze necessarie sono già integrate. Inoltre, per sfruttare a pieno la presenza necessaria di *tshark*, è stata sviluppata la possibilità di cattura e successiva traduzione automatica dei pacchetti nella rete.

7.1 Sviluppi futuri

In futuro, si prevede di perfezionare l'analisi dei pacchetti aggiungendo il supporto per nuovi protocolli e l'immissione di nuovi campi, in modo tale da permettere a *Google SecOps* l'individuazione di ulteriori attacchi.

Oltre alle aggiunte suggerite durante la stesura del *report*, un altro possibile approfondimento può riguardare alcune tipologie di pacchetti del livello applicativo (ad esempio, *HTTP*), i quali potrebbero contenere strutture peculiari, come nel caso di *WebDAV*. Per questo motivo sarebbe ottimo mostrare un atteggiamento più flessibile, così da permettere ai clienti di integrare moduli specifici per la decifrazione del traffico specifico delle loro applicazioni.

Non avendo accesso diretto a *Google SecOps*, che è disponibile per aziende con un contratto *Google Cloud Enterprise* attivo, si è testata l'applicazione solo a livello sperimentale e non in un ambiente operativo reale. Si sono quindi concentrati gli sforzi nella realizzazione della parte "*client*" sviluppando un *container docker* pronto all'uso diretto.

Per quanto riguarda quest'ultima parte, l'esecuzione dello *script entripoint* presenta potenziali miglioramenti per ridurre il carico *CPU* e aumentare la resistenza a possibili *fault*. Ad esempio, si potrebbe implementare un sistema di *message broker* per la comunicazione tra *tshark* e lo *script Python*.

Infine, si potrebbe introdurre un *database* per la gestione dei *log*, adattando la rotazione dei file in scrittura per garantirne una migliore integrazione.

7.2 Considerazioni finali

Con questo progetto, siamo riusciti a integrare le elevate capacità di uno strumento come *Wireshark*, in grado di catturare e analizzare in dettaglio il traffico di rete, con le funzionalità analitiche avanzate offerte da *Google SecOps*. Questa combinazione ci ha permesso di ottenere il meglio di entrambi i mondi, creando un *IDS* efficace e completo.

Sitografia

- [1] *Google Security Operations. Google Cloud.* 2024. URL: <https://cloud.google.com/security/products/security-operations?hl=it> (visitato il giorno 22/01/2025).
- [2] *PCAP man page. tcpdump.* 2024. URL: <https://www.tcpdump.org/manpages/pcap.3pcap.html> (visitato il giorno 22/01/2025).
- [3] *TShark. Terminal-Based Wireshark.* 2025. URL: [https://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html#:~:text=TShark%20is%20a%20terminal%20oriented,tshark%20\)%20or%20the%20online%20version.](https://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html#:~:text=TShark%20is%20a%20terminal%20oriented,tshark%20)%20or%20the%20online%20version.) (visitato il giorno 22/01/2025).
- [4] *Ingestion API. Google Security Operations.* 2024. URL: <https://cloud.google.com/chronicle/docs/reference/ingestion-api> (visitato il giorno 22/01/2025).
- [5] *UDM. Modello di dati unificato.* 2024. URL: <https://cloud.google.com/chronicle/docs/event-processing/udm-overview?hl=it> (visitato il giorno 22/01/2025).
- [6] *OWASP. Open Source Foundation for Application Security.* 2025. URL: <https://owasp.org> (visitato il giorno 27/01/2025).
- [7] *CVE. Security vulnerability database.* 2024. URL: <https://www.cvedetails.com/> (visitato il giorno 28/01/2025).
- [8] *OWASP Top Ten. 2021.* 2025. URL: <https://owasp.org/www-project-top-ten/> (visitato il giorno 28/01/2025).
- [9] *Formattazione come UDM. Google Security Operations.* 2024. URL: <https://cloud.google.com/chronicle/docs/unified-data-model/format-events-as-udm?hl=it> (visitato il giorno 22/01/2025).
- [10] *ISO 8601. Formato standard internazionale data e ora.* 2024. URL: <https://www.iso.org/iso-8601-date-and-time-format.html> (visitato il giorno 26/01/2025).
- [11] *Pyshark. libreria python.* 2024. URL: <https://pyshark-packet-analysis.readthedocs.io/en/latest/> (visitato il giorno 23/01/2025).
- [12] *Scapy. libreria python.* 2024. URL: <https://scapy.net/> (visitato il giorno 24/01/2025).
- [13] *Dpkt. libreria python.* 2024. URL: <https://dpkt.readthedocs.io/en/latest/> (visitato il giorno 25/01/2025).
- [14] *google-auth. Google API Client Library for Python.* 2024. URL: <https://google-auth.readthedocs.io/en/latest> (visitato il giorno 25/01/2025).
- [15] *dockerfile. Docker docs.* 2024. URL: <https://docs.docker.com/reference/dockerfile/> (visitato il giorno 30/01/2025).
- [16] *YAML. red hat: What is YAML?* 2024. URL: <https://www.redhat.com/en/topics/automation/what-is-yaml> (visitato il giorno 28/01/2025).
- [17] *compose-volumes. Docker docs.* 2024. URL: <https://docs.docker.com/engine/storage/volumes/> (visitato il giorno 29/01/2025).
- [18] *compose-envs. Docker docs.* 2024. URL: <https://docs.docker.com/compose/how-tos/environment-variables/set-environment-variables/> (visitato il giorno 29/01/2025).
- [19] *inotify. Linux manual page.* 2024. URL: <https://man7.org/linux/man-pages/man7/inotify.7.html> (visitato il giorno 28/01/2025).