

Capitolo 3

Progettazione ed Implementazione

3.1 Descrizione del problema

Il problema affrontato in questa tesi riguarda l’automazione delle risposte agli attacchi informatici. In particolare, si analizza la possibilità di automatizzare il deployment delle contromisure difensive, nello specifico, gli agenti Blue di **CALDERA** in base agli alert generati dal sistema di rilevamento **Wazuh**, all’interno dell’ambiente emulato **KathaRange**.

Allo stato attuale, **CALDERA** consente il deployment degli agenti Blue soltanto in modalità manuale, come reazione ad azioni offensive simulate da un agente Red. L’obiettivo principale della tesi è dunque quello di rendere automatico questo processo. Nello scenario ideale, quando un **IDS** (Intrusion Detection System) genera un alert a seguito di un’azione svolta da un agente Red, il sistema dovrebbe attivare automaticamente il corrispondente agente Blue, configurato con il profilo appropriato per contrastare la minaccia identificata.

Questa tesi propone una soluzione software sviluppata in Python per automatizzare tale processo, consentendo l’integrazione efficace tra il sistema **Wazuh** e **CALDERA**, al fine di garantire risposte rapide e automatiche agli attacchi informatici.

3.2 Soluzione Proposta

La soluzione proposta ha previsto l’utilizzo dell’ambiente simulato **KathaRange**, piattaforma che consente di creare reti virtualizzate basate su container Docker, utili per attività di emulazione di attacchi e difese informatiche. In particolare, **KathaRange** supporta l’integrazione di **CALDERA**, piattaforma che permette di emulare tecniche e tattiche offensive definite secondo la matrice **MITRE ATT&CK**.

Nello scenario sviluppato, gli attacchi simulati dai Red Agents da **CALDERA** sono rilevati dagli agenti di **Wazuh**, il quale opera come componente di monitoraggio, raccogliendo, indicizzando e analizzando in tempo reale eventi di sicurezza provenienti dai vari endpoint.

La soluzione è stata sviluppata considerando l'architettura generale del sistema, o meglio i vincoli ed il contesto di sicurezza in cui deployare la risposta difensiva. Le componenti software sono state sviluppate in Python, con una modalità di integrazione tramite **API REST** sia con **CALDERA** sia con il **Wazuh Indexer**; i cui dettagli implementativi, architetturali e vincoli progettuali saranno descritti nel paragrafo successivo.

Più precisamente, la componente software, attraverso uno scheduling temporale, interroga l'indice degli alert di sicurezza di **Wazuh** tramite richieste **HTTP GET**, sfruttando le **API** esposte. I risultati vengono analizzati mediante un processo di parsing del file **JSON**, allo scopo di identificare la tecnica **MITRE ATT&CK** utilizzata durante l'attacco simulato.

Identificata la tecnica offensiva associata all'alert, la componente Python invoca automaticamente, mediante chiamate **HTTP POST** alle **API REST** di **CALDERA** (in particolare sfruttando l'endpoint relativo alle Operations), la creazione e l'attivazione di un agente difensivo (Blue Agent) con il profilo di adversary opportuno. Tale agente, una volta attivato, esegue automaticamente le contromisure specifiche per contrastare la tecnica offensiva rilevata.

Questo flusso automatizzato consente una reazione rapida e precisa agli attacchi simulati, realizzando un processo integrato di detection and response all'interno dell'ambiente virtualizzato **KathaRange**.

3.3 Architettura del sistema

La progettazione architetturale riguarda la comprensione di come dovrebbe essere organizzato un sistema software e la definizione della sua struttura complessiva. Essa costituisce un collegamento fondamentale tra la progettazione e i requisiti, in quanto identifica i principali componenti strutturali di un sistema e le relazioni tra essi. Il risultato di questa fase è un modello architetturale che descrive come il sistema sia organizzato come insieme di componenti che comunicano tra loro.

La soluzione architetturale individuata ha tenuto conto della modalità di esposizione dei servizi, delle operazioni di attacco simulate e dei componenti di seguito elencati:

- **Wazuh server:** RESTful API
- **Wazuh Indexer:** RESTful API
- **Data Storage di Wazuh alert**
- **Comunicazione crittografata:** Autenticazione con username e password
- **Formato dei dati persistiti:** JSON
- **CALDERA:** RESTful API
- **CALDERA Operations:** Blue e Red

L'architettura del sistema implementato si basa sull'ambiente simulato e containerizzato **KathaRange**, che emula un'infrastruttura di cybersicurezza ed è composta da tre nodi principali, descritti di seguito.

3.3.1 Docker Container per Blue Deploy

Container che ospita lo script `Blue_Deploy.py`, sviluppato in Python, che rappresenta il core del sistema automatizzato di risposta. Le funzioni principali implementate sono:

- Monitorare gli alert generati dal sistema di detection **Wazuh Indexer**.
- Analizzare gli alert per identificare le tecniche di attacco **MITRE ATT&CK**.
- Inviare richieste **HTTP** alle **API REST** di **CALDERA** per attivare automaticamente il Blue Agent.
- Comunicare con il server **CALDERA** per attivare automaticamente le contromisure (Blue Operation).

3.3.2 Wazuh Server Container

Questo nodo rappresenta il sistema **SIEM** (Security Information and Event Management) e include:

- **Wazuh API**: espone endpoint **REST** per interrogare lo stato dei nodi monitorati e gli alert.
- **Wazuh Indexer**: indicizza e archivia gli eventi di sicurezza generati da Wazuh.

3.3.3 Caldera Server Container

Questo nodo è responsabile della simulazione e gestione delle risposte difensive e ospita:

- **CALDERA REST API**: consente la creazione e l'esecuzione di operazioni difensive via **HTTP**.
- **Blue Agent**: un agente difensivo che può essere avviato in risposta ad alert specifici.
- **Operation Handler**: componente incaricato di coordinare e avviare le contromisure automatizzate in base alla tecnica rilevata.

3.3.4 Flusso di Comunicazione

Il flusso complessivo si svolge come segue:

1. Lo script Python interroga **Wazuh Indexer** per rilevare nuovi alert.
2. Analizza il contenuto degli alert per individuare la tecnica di attacco.
3. Se rilevante, invia una richiesta alle **API** di **CALDERA** per attivare un Blue Agent con il profilo difensivo adeguato.
4. Il sistema **CALDERA** avvia automaticamente l'operazione, simulando la risposta a un attacco informatico.

Tutte le comunicazioni tra i componenti avvengono tramite **API REST** protette (autenticazione e, se necessario, **TLS**), su una rete interna simulata tramite **Katharange**.

In particolare, le componenti di riposta automatica sviluppate, deployate all'interno di un Docker container, ognuna con un compito specifico, sono di seguito descritte:

- **wazuh_api.py**: componente deputata ad interrogare attraverso le API il **Wazuh Indexer** per ottenere i log degli attacchi rilevati.
- **caldera_api.py**: componente che ha il compito di invocare le **API** di **CALDERA** per la gestione di agenti, abilità e operazioni.
- **Blue_Deploy.py**: è la componente principale che gestisce la pipeline di risposta automatica.

L'architettura complessiva del sistema realizzato è rappresentato nella figura seguente:

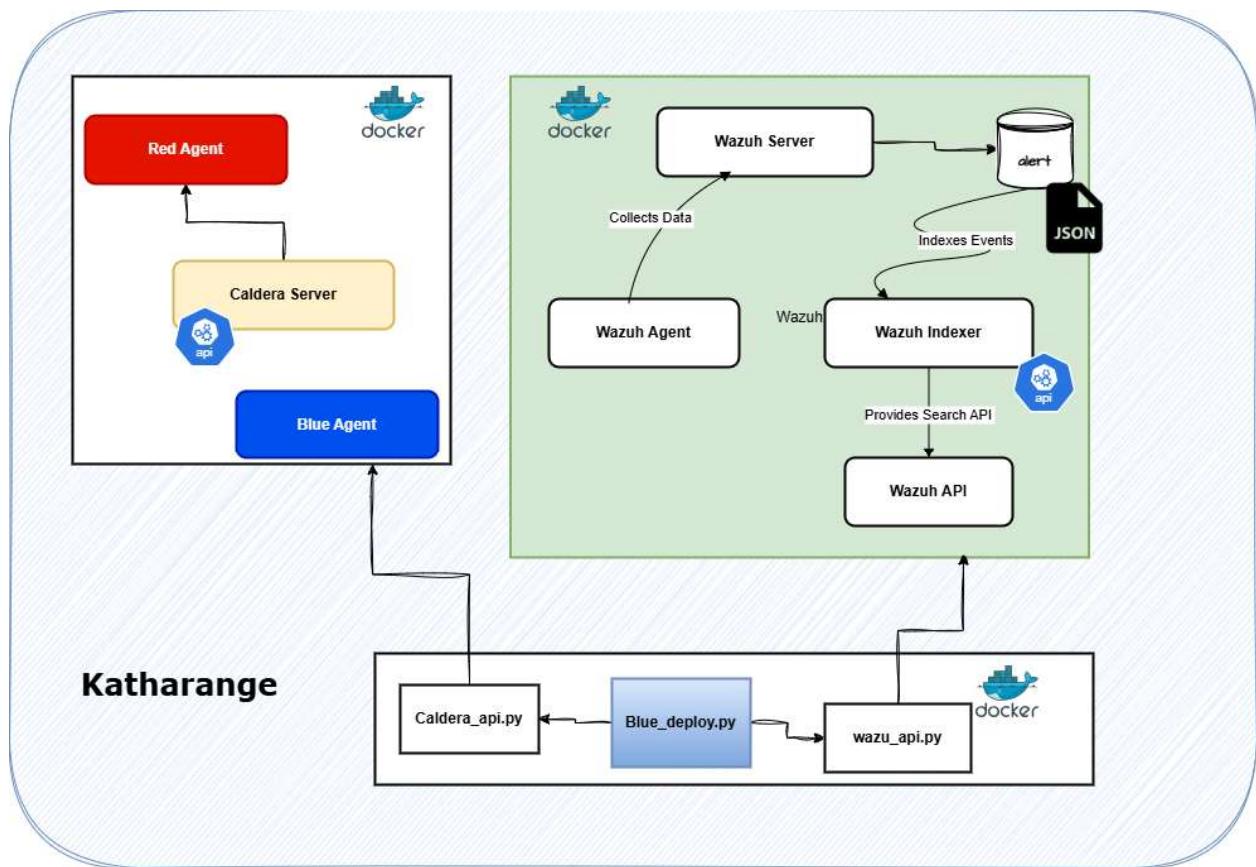


Figura 3.1: Architettura del sistema

3.4 Logica di funzionamento

La logica di funzionamento al fine di migliorare sia la leggibilità che la comprensibilità di quanto realizzato è stato descritta con l'ausilio del diagramma di flusso rappresentato in figura 3.2 e con il sequence diagram in figura 3.3 che descrive lo scenario del deployment automatico del Blue Agent.

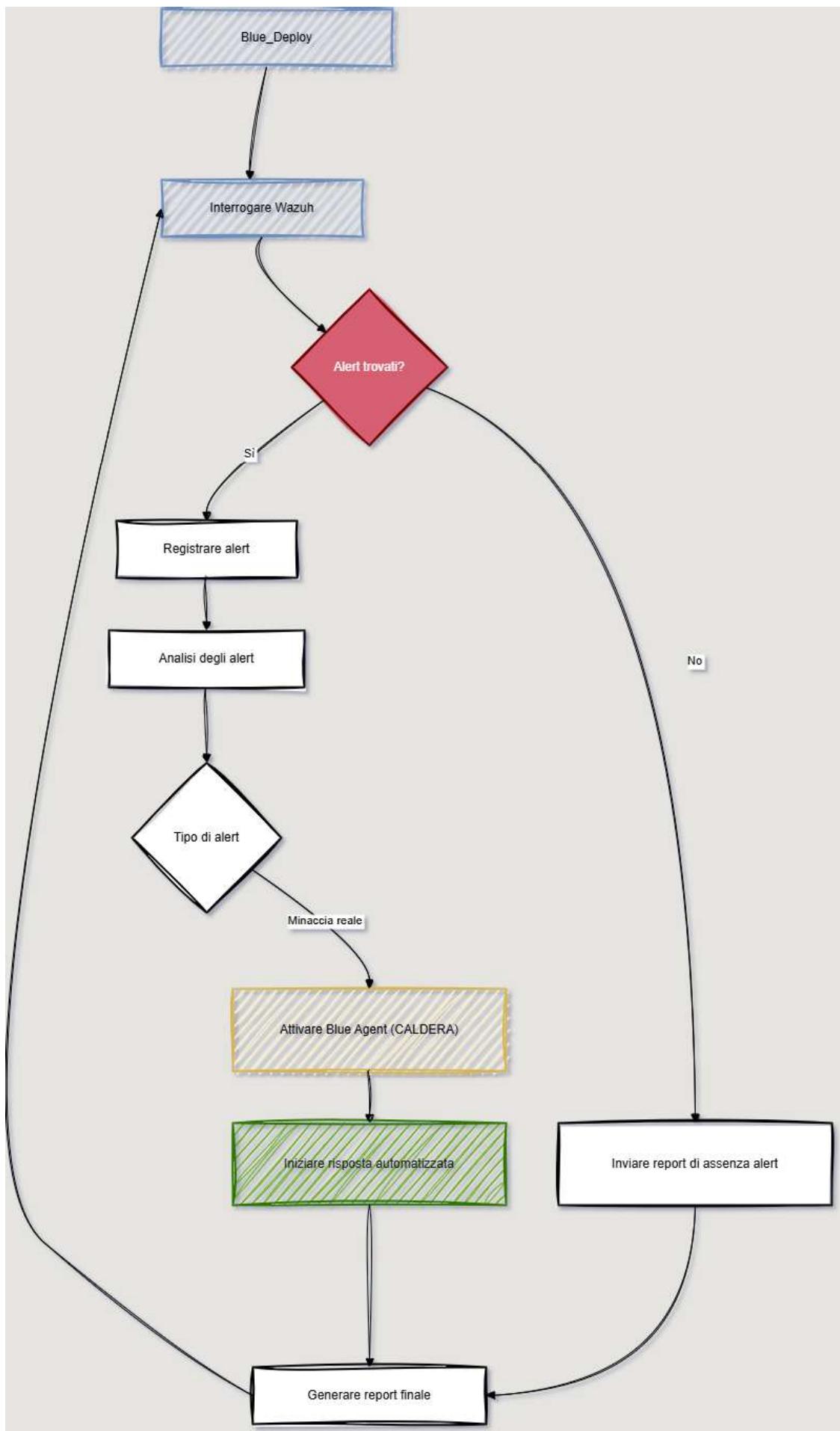


Figura 3.2: Diagramma di Flusso della logica di funzionamento

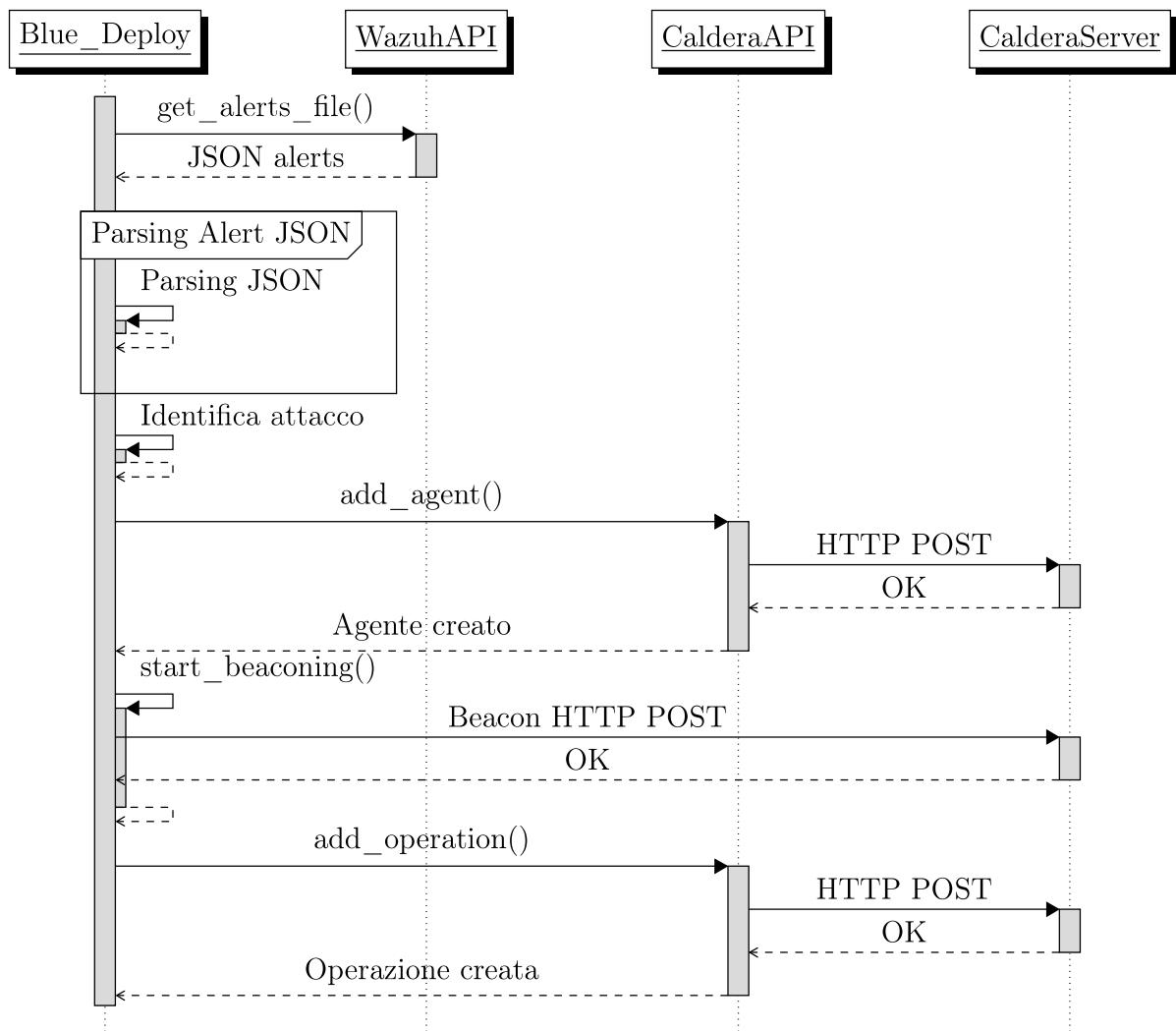


Figura 3.3: Sequence Diagram del deployment automatico del Blue Agent

3.4.1 Ottenimento File Minacce

La funzione principale di **Blue_Deploy.py** avvia un ciclo infinito che, utilizzando la funzione `get_alerts_file()` interroga l'istanza di **Wazuh** generando il file `syscheck_events.json`. Il file contiene le minacce rilevate ordinate per timestamp, dalle quali si estrae l'**ID** della tecnica **MITRE** utilizzata nell'attacco.

```
1 print("\nGetting Alert Events:")
2     response = requests.get(url, auth=auth, verify=False)
3     # Copia i risultati in un file
4     with open('syscheck_events.json', 'w') as f:
5         f.write(response.text)
```

Figura 3.4: Ottenimento file minacce

```
1 while(True):
2     wazuh.get_alerts_file()
3     with open('syscheck_events.json', 'r') as f:
4         data = json.load(f)
```

Figura 3.5: Controllo file delle minacce

```
1} syscheck_events.json > {} hits > [ ]hits > {} 0 > {} _source > {} rule
10    "hits" : [
11      "hits" : [
12        {
13          "_source" : {
14            "rule" : {
15              ],
16              "tsc" : [
17                "CC6.8",
18                "CC7.2",
19                "CC7.3"
20              ],
21              "description" : "Root's crontab entry changed.",
22              "groups" : [
23                "syslog",
24                "cron"
25              ],
26              "nist_800_53" : [
27                "AU.14",
28                "AU.6",
29                "AC.6"
30              ],
31              "gdpr" : [
32                "IV_35.7.d",
33                "IV_32.2"
34              ],
35              "firedtimes" : 1,
36              "mitre" : {
37                "technique" : [
38                  "Cron"
39                ],
40                "id" : [
41                  "T1053.003"
42                ],
43                "tactic" : [
44                  "Execution",
45                  "Persistence",
46                  "Privilege Escalation"
47                ]
48              }
49            }
50          ]
51        }
52      ]
53    ]
54  ]
55 ]
56 }
```

Figura 3.6: Contenuto File syscheck_event.json

3.4.2 Controllo Presenza Nuovo Attacco

Dopo aver ottenuto i log più recenti da **Wazuh**, il sistema estrae il campo ***timestamp*** riguardante l'evento più recente. Questo ***timestamp*** viene confrontato con l'ultimo ***timestamp*** salvato (variabile ***old***) per determinare se l'attacco è nuovo oppure già noto. Se il ***timestamp*** attuale è più recente rispetto a quello nella variabile ***old*** allora l'evento viene considerato come un nuovo attacco e vengono prese le adeguate contromisure. In caso contrario, il ciclo continua in attesa di un nuovo evento, senza eseguire alcuna azione.

```

1 timestamp = data['hits']['hits'][0]['_source']['predecoder'][
2     'timestamp']
3 now = datetime.strptime(timestamp, date_format)
4 old = datetime.strptime(old, date_format)
5
6 if now > old:
7     old = timestamp
8     print(f"Red agent attack detected")
9     # avvio agente blue
10 else:
11     old = timestamp
12     print(f"No new attacks detected")

```

Figura 3.7: Controllo della presenza di un nuovo attacco

3.4.3 Identificazione Tecnica MITRE

Dopo aver rilevato un nuovo attacco, il sistema analizza l'evento ricavato da **Wazuh** per poter ottenere l'**ID** della tecnica **MITRE ATT&CK** utilizzata nell'attacco. Successivamente si utilizza questo **ID** per individuare la contromisura adeguata.

La funzione ***get_required_response()***, tramite un dizionario, effettua questa mappatura, restituendo nella variabile ***adversary_blue*** l'**ID** di un profilo avversario preconfigurato in **CALDERA**, contenente le abilità difensive necessarie per difendersi dall'attacco.

```

1 id = data['hits']['hits'][0]['_source']['rule']['mitre']['id']
2 print(f"Attack Tactic ID: {id[0]}")
3 adversary_blue = get_required_response(id[0])

```

Figura 3.8: Identificazione dell'**ID** MITRE

```

1 def get_required_response(adversary_id):
2     # Mappa le tecniche MITRE agli adversary profile di risposta
3     abilities_mapping = {
4         "T1053.003": "169cdc73-8fea-49cf-9021-d0b3c24e2b17",
5         "T1136": "169cdc73-8fea-49cf-9021-d0b3c24e2b17"
6     }
7     return abilities_mapping.get(adversary_id, [])

```

Figura 3.9: Funzione di mappatura tra tecnica MITRE e profilo di risposta difensivo.

3.4.4 Deployment Blue Agent

Dopo aver individuato la contromisura più appropriata, il sistema crea e attiva un agente **CALDERA** appartenente al gruppo "blue", incaricato di eseguire l'operazione di risposta. Il blue agent invierà peridicamente dei segnali beacon per notificare la sua presenza a **CALDERA**.

```

1 blue_agent = Agent(group = "blue", key = CALDERA_API_KEY_BLUE)
2 blue_agent.start_beaconing()

```

Figura 3.10: Attivazione e avvio del beaconing dell'agente blue.

Agents								
You must deploy at least 1 agent in order to run an operation. Groups are collections of agents so hosts can be compromised simultaneously.								
+ Deploy an agent		Configuration		1 alive 1 trusted 1 agent 0 dead 0 untrusted				Bulk Actions
id (paw)	host	group	platform	contact	pid	privilege	status	last seen
mxhdvn	vinz-HP-Pavilion-Sleekbook-15	blue	linux	http	1	Elevated	alive, trusted	17/04/2025, 18:08:27

Figura 3.11: Agente Blue attivo su CALDERA

3.4.5 Classe CALDERA Agent

Il processo di beaconing è gestito in un thread separato per permettere al blue agent di eseguire più operazioni contemporaneamente senza bloccare l'esecuzione del programma. La funzione `start_beaconing()` crea un thread dedicato che effettua il beaconing ogni `beaconing_interval`. Ogni volta che si attiva il thread, il beacon viene inviato tramite la funzione `send_beacon()` con i dati necessari, al server **CALDERA**. Il beaconing viene gestito tramite la libreria `threading` di Python. Per deployare un agente su **CALDERA** è necessario effettuare una **HTTP POST** contenente una struttura **JSON** che rispecchia quella richiesta dall'**API REST** di **CALDERA**. Questa operazione viene svolta dalla funzione `add_agent()`.

```

1 def add_agent(self, name, group, platform = platform.system().
2     lower(),
3 paw = ''.join(random.choices(string.ascii_lowercase, k=6)),
4 architecture = "amd64", pid = os.getpid(), ppid = os.getppid(),
5 host = socket.gethostname(), sleep_min=30, sleep_max=60,
6 watchdog=0,
7 location = "/home/vinz/KathaRange/lab/sandcat.go-linux",
8 trusted=True, pending_contact="HTTP", privilege="Elevated",
9 host_ip_addrs=socket.gethostbyname(socket.gethostname())):
10
11     data = {
12         "paw": paw,
13         "sleep_min": sleep_min,
14         "sleep_max": sleep_max,
15         "watchdog": watchdog,
16         "group": group,
17         "architecture": architecture,
18         "platform": platform,
19         "server": "http://0.0.0.0:8888",
20         "upstream_dest": "http://0.0.0.0:8888",
21         "username": username,
22         "location": location,
23         "pid": pid,
24         "ppid": ppid,
25         "trusted": trusted,
26         "executors": [
27             "proc",
28             "sh"
29         ],
30         "privilege": privilege,
31         "exe_name": name,
32         "host": host,
33         "contact": "HTTP",
34         "proxy_receivers": {
35

```

Figura 3.12: Funzione add_agent().

```

1      "proxy_chain": [
2
3        ],
4
5      "origin_link_id": "string",
6      "deadman_enabled": True,
7      "available_contacts": [
8        "HTTP"
9      ],
10     "host_ip_addrs": [
11       host_ip_addrs
12     ],
13     "pending_contact": pending_contact
14   }
15
16   if self.PRINT_DEBUG:
17     print(f"URL: {self.AGENTS_ENDPOINT}")
18
19   try:
20     data = self._make_web_request(self.AGENTS_ENDPOINT,
21       body=data, method='POST')
22     return paw
23   except Exception as e:
24     print(self._error_message(e))

```

Figura 3.13: Funzione add_agent() 2.

```

1 class Agent:
2
3   def __init__(self, group: str, key: str, paw: str | None =
4     None, beaconing_interval: int = 5, caldera_instance: api.
5     Caldera | None = None):
6
7     self.beaconing: bool = False
8     self.beaconing_interval: int = beaconing_interval
9     self.caldera: api.Caldera = api.Caldera(key, debug=True,
10       print_banner=False) if not caldera_instance else
11       caldera_instance
12     self.paw: str = paw if paw else self.caldera.add_agent(
13       name="sandcat.go-linux", group=group)
14     self.group: str = group

```

Figura 3.14: Classe per creare agenti CALDERA

```

1 def start_beaconing(self) -> None:
2     if self.beaconing:
3         print("Beaconing already started")
4         return
5     self.beaconing = True
6     self.beaconing_thread = threading.Thread(target=self.
7         beacon, daemon=True)
8     self.beaconing_thread.start()
9
10
11
12
13
14
15
16
17 def beacon(self) -> None:
18     while self.beaconing:
19         # Invia un beacon al server
20         response = self.send_beacon()
21         if response.status_code != 200:
22             raise Exception(f"Failed to send beacon: {response.
23                             text}")
24         time.sleep(self.beaconing_interval)
25
26
27 def send_beacon(self) -> requests.Response:
28     data_dict = {
29         "paw": self.paw,
30         "server": self.caldera.caldera_URL,
31         "pid": os.getpid(),
32     }
33     # Converto il dizionario in JSON e lo codifico in base64
34     json_data = json.dumps(data_dict)
35     base64_data = base64.b64encode(json_data.encode('utf-8')).decode('utf-8')
36     # Invio il beacon al server
37     response = requests.post(self.caldera.caldera_URL + '/beacon',
38         data=base64_data)
39     return response

```

Figura 3.15: Funzione beaconing dell'agente blue.

3.4.6 Attivazione Contromisure

Dopo aver attivato il blue agent, il sistema avvia l'operazione di difesa automatica tramite l'API di **CALDERA**. L'operazione da avviare viene selezionata utilizzando la variabile *adversary_blue*, valorizzata come descritto nella fase precedente, ed assegnata al gruppo "blue". Ogni operazione viene nominata in modo incrementale (Blue Response 1, Blue Response 2, ecc.) per facilitare la tracciabilità nel tempo. La funzione *add_operation()* dell'API di caldera effettua una **HTTP POST** e crea l'operazione.

```

1 op_name = f"Blue Response {number}"
2 operation = blue_agent.caldera.add_operation(name=op_name, group="blue",
3                                                 adversary_id=adversary_blue
4 )

```

Figura 3.16: Avvio automatico dell'operazione difensiva

The screenshot shows the CALDERA Operations interface. At the top left, it says "Operations". Below that, a section titled "Blue Response 1" contains a "Download Graph SVG" button. To the right, there's a dropdown menu with "Blue Response 1 - 1 decisions | 7 min ago" and a "New Operation" button. A "Download Report" button is also visible. The main area lists several operations: "wazuh_JDS_Test - 2 decisions | 8 min ago", "Blue Response 1 - 1 decisions | 7 min ago" (which is highlighted in blue), and "Blue Response 2 - 1 decisions | 6 min ago". At the bottom center is a red penguin icon with the text "vinz-HP-Pavilion-Sleekbook-15".

Figura 3.17: Operazioni Blue attive su CALDERA

3.5 Vantaggi dell'Architettura

Le componenti sviluppate introducono un vantaggio significativo in termini di automazione del processo di risposta agli attacchi informatici. Il sistema è infatti in grado di reagire in tempo reale agli alert di sicurezza, deployando automaticamente le contromisure più appropriate in base alla tecnica **MITRE ATT&CK** rilevata. Questo approccio consente di ridurre sensibilmente i ritardi decisionali dovuti alla necessità di un intervento umano, minimizzando il rischio di errori operativi e migliorando l'efficienza e l'efficacia complessiva della risposta.

L'adozione di meccanismi di autenticazione, come token, credenziali **HTTP Basic** o **API key**, garantisce che soltanto le entità autorizzate possano accedere agli endpoint esposti. Ogni request è quindi tracciabile e attribuibile a un soggetto verificabile, inoltre le interfacce RESTful permettono di definire in modo preciso e limitato i punti di accesso al sistema, riducendo la superficie di attacco.

Dal punto di vista della protezione dei dati in transito, le **API REST** si integrano facilmente con protocolli sicuri come **HTTPS**, che, grazie alla crittografia **TLS** (Transport Layer Security), assicurano la riservatezza e l'integrità delle comunicazioni. In questo modo, informazioni sensibili come identificatori di attacco, parametri di configurazione o riferimenti a profili **MITRE ATT&CK** non possono essere intercettate o modificate da soggetti non autorizzati.