Network Cybersecurity Intrusion System: Thesis Draft

Gaetano Celentano

August 21, 2025

Contents

Chapter 1

Introduction

This thesis presents the design and implementation of a Network Cybersecurity Intrusion System (NCIS). The project aims to detect, monitor, and mitigate network attacks using a modular Python-based architecture. La presente tesina illustra il percorso di progettazione, sviluppo e validazione di un sistema modulare per la sicurezza delle reti SDN, denominato Network Cybersecurity Intrusion System (NCIS). L'obiettivo principale è quello di realizzare una soluzione capace di rilevare, monitorare e mitigare attacchi di rete in tempo reale, con particolare attenzione alla flessibilità, all'estendibilità e all'aderenza agli scenari reali. Il lavoro si inserisce nel contesto della crescente adozione di architetture SDN, che richiedono strumenti di sicurezza evoluti e adattivi. La trattazione segue un approccio critico, evidenziando le scelte progettuali e le motivazioni che hanno guidato lo sviluppo, con riferimento costante alle problematiche riscontrate nelle soluzioni precedenti.

Chapter 2

Requisiti, Struttura e Decisioni Progettuali

2.1 Miglioramenti rispetto alla soluzione precedente

Il progetto NCIS nasce dalla volontà di superare i limiti riscontrati nella soluzione presentata lo scorso anno. Di seguito si illustrano i principali difetti individuati e le relative correzioni implementate, con una riflessione sul loro impatto.

1. Over blocking: In passato, il sistema bloccava intere porte degli switch, causando la perdita di traffico legittimo e penalizzando utenti non coinvolti negli attacchi. La nuova implementazione introduce un blocco a livello di flusso e indirizzo MAC, supportato da whitelist e blocklist, riducendo drasticamente l'impatto sugli utenti legittimi e rendendo la mitigazione più precisa.

2. Static threshold: Le soglie di rilevamento erano fissate staticamente sulla topologia, rendendo il sistema poco flessibile e incline a errori in presenza di variazioni di traffico. Ora sono state adottate soglie adattive basate su medie mobili e percentili, che si adattano dinamicamente al traffico, migliorando la capacità di rilevare attacchi senza generare falsi positivi.

3. Mancanza di modularità: La precedente soluzione accorpava monitoraggio, decisione ed enforcement in un unico blocco di codice, rendendo difficile la manutenzione e l'estensione. L'architettura attuale è modulare, con thread separati per monitoring, policy computation ed enforcement, favorendo la chiarezza, la manutenibilità e la scalabilità del sistema.

4. Detection limitata a DoS classici: Il sistema rilevava solo attacchi UDP a bitrate costante, risultando inefficace contro pattern più sofisticati. Ora sono gestiti anche attacchi bursty e distribuiti, grazie all'introduzione di metriche aggiuntive come la varianza del traffico e l'analisi degli intervalli tra i pacchetti, rendendo la detection più robusta e realistica.

5. Policy di blocco/sblocco inflessibile: I tempi di blocco erano troppo rigidi, con rischio di sblocco prematuro o penalizzazione eccessiva. L'adozione di strategie di backoff esponenziale e sblocco progressivo consente una gestione più flessibile e intelligente delle policy, riducendo sia i falsi positivi sia il rischio di rientro degli attaccanti.

Questi miglioramenti sono stati introdotti per rendere il sistema più aderente alle esigenze di sicurezza di una rete SDN reale, aumentando la precisione, la resilienza e la capacità di adattamento del progetto.

2.2 Requisiti

Il sistema è stato progettato per soddisfare i seguenti requisiti fondamentali:

- **Rilevamento in tempo reale**: capacità di individuare attacchi di rete con latenza minima, garantendo una risposta tempestiva.
- **Mitigazione automatica**: applicazione di contromisure efficaci per limitare l'impatto degli attacchi, con particolare attenzione alla minimizzazione dei falsi positivi.
- **Monitoraggio e logging**: raccolta continua di statistiche e eventi di rete, con registrazione dettagliata delle azioni intraprese per facilitare analisi forensi e tuning.
- **Interfaccia REST**: esposizione di endpoint per l'interazione con il sistema, favorendo l'integrazione con dashboard, strumenti di orchestrazione e automazione.
- **Modularità ed estendibilità**: possibilità di aggiungere facilmente nuovi moduli o algoritmi, adattando il sistema a scenari e minacce emergenti.

Questi requisiti sono stati definiti sulla base delle criticità riscontrate nelle soluzioni precedenti e delle best practice per la sicurezza in ambienti SDN.

In prospettiva, il sistema potrà essere esteso per includere funzionalità di analisi predittiva, integrazione con sistemi di threat intelligence e supporto a protocolli di orchestrazione avanzata.

2.3 Design e Struttura del Progetto

L'architettura del sistema è fortemente modulare: ogni componente è responsabile di un compito specifico e comunica con gli altri tramite chiamate di funzione e endpoint REST. Questa scelta consente di isolare le logiche di detection, mitigazione e monitoraggio, facilitando la manutenzione e l'aggiornamento del sistema. La modularità è stata ulteriormente rafforzata dall'adozione di thread separati per le principali attività, riducendo i colli di bottiglia e migliorando la scalabilità.

La progettazione ha tenuto conto anche della necessità di adattarsi a topologie di rete diverse e a pattern di traffico variabili, prevedendo la possibilità di configurare facilmente le metriche di detection e le policy di risposta. L'approccio seguito permette di validare il sistema sia in ambienti di test controllati sia in scenari più complessi e realistici.

Questa flessibilità architetturale rappresenta un punto di forza per l'evoluzione futura del progetto.

La struttura del progetto NCIS riflette una precisa scelta architetturale volta a garantire modularità, manutenibilità e chiarezza. Il codice è suddiviso in moduli distinti, ciascuno responsabile di un aspetto fondamentale del sistema: l'API REST (`api.py`) consente l'interazione esterna e l'integrazione con altri strumenti; il controller (`controller.py`) coordina le attività di detection e mitigazione; il modulo di rilevamento (`detector.py`) implementa le logiche adattive per l'identificazione degli attacchi; il mitigatore (`mitigator.py`) gestisce le strategie di risposta e blocco; il monitor (`monitor.py`) raccoglie e aggiorna le statistiche di rete necessarie per la detection; infine, i file di topologia (`topology.py`, `topology_new.py`) permettono di simulare diversi scenari di rete per la validazione e il testing.

In aggiunta, la presenza di file di documentazione e test (`Proceedings.md`, `Tests.md`) garantisce la tracciabilità delle decisioni progettuali e la riproducibilità dei risultati, elementi fondamentali in un contesto accademico e professionale. La struttura modulare consente inoltre di integrare facilmente nuovi algoritmi di detection o strategie di mitigazione, favorendo la sperimentazione e l'aggiornamento continuo.

Questa organizzazione è stata pensata per facilitare la collaborazione tra sviluppatori, la revisione da parte di terzi e l'eventuale trasferimento tecnologico verso ambienti produttivi.

2.4 Decisioni Progettuali

Durante lo sviluppo del progetto sono state prese numerose decisioni architetturali e implementative, documentate nei file di proceedings. Questi documenti tracciano il percorso progettuale, le motivazioni delle scelte e le eventuali alternative considerate, fornendo trasparenza e facilitando la collaborazione tra diversi membri del team o la revisione da parte di terzi.

La presenza di una documentazione strutturata delle decisioni è un elemento distintivo del progetto, che ne aumenta la qualità e la professionalità.

Il progetto NCIS è stato sviluppato seguendo un approccio modulare, con la suddivisione delle responsabilità tra i diversi file principali. Il controller gestisce l'orchestrazione dei moduli e gli eventi OpenFlow, avviando thread dedicati per monitoraggio, detection, mitigazione e API REST. Il monitor raccoglie periodicamente statistiche granulari dagli switch, fornendo dati aggiornati per la detection. Il detector analizza le statistiche tramite plugin, utilizzando soglie adattive per rilevare anomalie e notificando il mitigator. Il mitigator applica regole di blocco e sblocco granulari, gestendo lo sblocco progressivo e il logging delle decisioni. L'API espone endpoint REST per la gestione dinamica delle regole.

Le scelte architetturali sono state guidate dalla necessità di garantire modularità, estendibilità e reattività. L'uso di thread separati assicura monitoraggio e risposta continua, mentre le soglie adattive e il blocco granulare riducono i falsi positivi e l'impatto sul traffico legittimo. La documentazione delle decisioni e la presenza di logging facilitano audit, debugging e future estensioni.

La soluzione è stata progettata per essere facilmente estendibile: nuovi plugin di detection, criteri di mitigazione e endpoint API possono essere aggiunti senza modificare la struttura esistente. Il sistema è pronto per ulteriori sviluppi e integrazioni, mantenendo la flessibilità necessaria per adattarsi a scenari e minacce emergenti.

Chapter 3

Implementation

In questa sezione vengono presentati i principali moduli implementativi del sistema, con particolare attenzione alle scelte tecniche e alle motivazioni che hanno guidato lo sviluppo. Ogni modulo è stato progettato per rispondere a specifiche esigenze di sicurezza e per superare i limiti delle soluzioni precedenti.

3.1 API

L'API REST rappresenta il punto di ingresso per tutte le interazioni esterne con il sistema. Attraverso endpoint dedicati, è possibile inviare richieste di detection, ricevere notifiche di allarme e consultare lo stato della rete. La scelta di una interfaccia RESTful garantisce interoperabilità, semplicità di integrazione e scalabilità, permettendo di collegare il sistema a dashboard di monitoraggio, orchestratori o altri strumenti di automazione.

L'implementazione dell'API è stata pensata per essere estendibile: nuovi endpoint possono essere aggiunti facilmente per supportare funzionalità avanzate o integrazioni future.

```
@app.route('/detect', methods=['POST'])
def detect():
    # Riceve dati, chiama il controller e restituisce il risultato
    result = controller.detect_attack(request.json)
    return jsonify(result)
```

3.2 Controller

Il controller costituisce il nucleo logico del sistema: si occupa di orchestrare le attività di monitoraggio, rilevamento e mitigazione, coordinando i diversi moduli in modo sinergico. La sua implementazione come classe centrale consente di gestire lo stato della rete, le statistiche raccolte e le policy di risposta in modo strutturato e flessibile.

La scelta di separare le responsabilità tra i moduli e di centralizzare la logica di coordinamento nel controller facilita la manutenzione, l'estensione e il debugging del sistema, rendendo più semplice l'integrazione di nuove funzionalità.

```
class ModularController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(ModularController, self).__init__(*args, **kwargs)
```

```
        self.mac_to_port = {}
        self.monitor = Monitor(self)
        self.detector = Detector(self)
        self.mitigator = Mitigator(self)
```

3.3 Detector

Il modulo di detection è stato progettato per superare i limiti delle soluzioni statiche, adottando algoritmi adattivi basati su medie mobili, deviazione standard e analisi dei pattern di traffico. Questa scelta consente di rilevare attacchi anche in presenza di variazioni dinamiche del traffico, riducendo sensibilmente i falsi positivi e aumentando la robustezza del sistema.

L'implementazione supporta la facile integrazione di nuovi algoritmi o metriche, favorendo la sperimentazione e l'aggiornamento continuo in risposta all'evoluzione delle minacce.

```
class AdaptiveThresholdPlugin(DetectionPlugin):
    def __init__(self, window=10, std_factor=3):
        self.window = window
        self.std_factor = std_factor
        self.history = {}
    def analyze(self, stats):
        anomalies = []
        for key, value in stats.items():
            throughput = value.get("throughput", 0)
            hist = self.history.setdefault(key, [])
            hist.append(throughput)
            if len(hist) > self.window:
                hist.pop(0)
            if len(hist) >= self.window:
                mean = np.mean(hist)
                std = np.std(hist)
                if throughput > mean + self.std_factor * std:
                    anomalies.append({"key": key, "throughput": throughput,
                    ↪  "mean": mean, "std": std})
        return anomalies
```

3.4 Mitigator

Il modulo Mitigator si occupa di applicare le strategie di risposta agli attacchi rilevati, intervenendo in modo automatico e granulare sui flussi sospetti. L'introduzione di meccanismi di whitelist, blocklist e backoff esponenziale consente di gestire le policy di blocco e sblocco in modo intelligente, minimizzando l'impatto sul traffico legittimo e riducendo il rischio di penalizzazioni ingiustificate.

Questa automazione è fondamentale per garantire una risposta tempestiva agli attacchi e per adattare il comportamento del sistema alle condizioni reali della rete.

```
class Mitigator:
    def handle_anomaly(self, anomaly):
```

```
        flow_id = anomaly.get("key")
        datapath = next(iter(self.controller.dps.values()), None)
        if datapath and flow_id:
            self.apply_block(datapath, flow_id)
            self.logger.info(f"Blocco automatico per anomalia: {flow_id}")
```

3.5 Monitor

Il modulo Monitor svolge un ruolo cruciale nella raccolta e nell'aggiornamento delle statistiche di rete, fornendo i dati necessari per la rilevazione tempestiva di anomalie e attacchi. La raccolta periodica delle metriche consente di mantenere una visione aggiornata dello stato della rete e di supportare le logiche adattive di detection.

La progettazione del monitor è stata pensata per essere estendibile: nuove metriche possono essere aggiunte facilmente per migliorare la capacità di analisi e la precisione del sistema.

```
class Monitor:
    def run(self):
        while self.running:
            self.collect_stats()
            time.sleep(self.interval)
    def collect_stats(self):
        for dp in getattr(self.controller, "dps", {}).values():
            parser = dp.ofproto_parser
            req = parser.OFPPortStatsRequest(dp, 0, dp.ofproto.OFPP_ANY)
            dp.send_msg(req)
            req = parser.OFPFlowStatsRequest(dp)
            dp.send_msg(req)
```

3.6 Topology

La validazione del sistema è stata effettuata tramite la simulazione di diverse topologie di rete, utilizzando ambienti di test sia semplici che complessi. L'ambiente di test ridotto consente di verificare il corretto funzionamento delle funzionalità di base, mentre la topologia complessa permette di stressare il sistema con traffico vario e attacchi multipli, testando la robustezza e la scalabilità delle soluzioni implementate.

Questa scelta metodologica garantisce che il sistema sia realmente efficace e adattabile in scenari SDN di diversa complessità, fornendo una base solida per futuri sviluppi e applicazioni in ambienti produttivi.

```
class Environment(object):
    def __init__(self):
        self.net = Mininet(controller=RemoteController, link=TCLink)
        self.h1 = self.net.addHost('h1', mac='00:00:00:00:00:01',
        ↪  ip='10.0.0.1')
        self.s1 = self.net.addSwitch('s1', cls=OVSKernelSwitch)
        self.net.addLink(self.h1, self.s1, bw=10)
        self.net.build()
        self.net.start()
```

```
class ComplexEnvironmentFixed(object):
    def __init__(self):
        self.net = Mininet(controller=RemoteController, link=TCLink)
        hosts = [self.net.addHost(f"h{i}", mac=f"00:00:00:00:00:0{i}",
        ↪  ip=f"10.0.0.{i}") for i in range(1,8)]
        switches = [self.net.addSwitch(f"s{i}", cls=OVSKernelSwitch) for i in
        ↪  range(1,11)]
        self.net.addLink(hosts[0], switches[0], bw=10)  # h1 -> s1
        self.net.addLink(switches[0], switches[6], bw=5)  # s1 -> s7
        self.net.build()
        self.net.start()
```

Chapter 4

Testing

La fase di testing ha rivestito un ruolo centrale nel processo di sviluppo: sono stati definiti e implementati casi di test specifici per validare la correttezza delle logiche di detection, la reattività delle strategie di mitigazione e la robustezza del sistema in presenza di traffico legittimo e malevolo.

I risultati dei test hanno permesso di affinare le soglie, le policy e le metriche utilizzate, garantendo un equilibrio tra efficacia della protezione e minimizzazione dei falsi positivi. La documentazione dei test facilita la riproducibilità degli esperimenti e la comparazione con soluzioni alternative.

4.1 Metodologia e scenari di test

La validazione del sistema NCIS è stata condotta tramite simulazioni in ambienti Mininet, utilizzando sia topologie semplici che complesse. Nella topologia semplice (3 host, 4 switch), si è verificata la capacità del controller di distinguere tra traffico DoS e traffico legittimo, generando attacchi flood UDP/TCP e traffico normale (ping, iperf). Nella topologia complessa (7 host, 10 switch), si è dimostrata la scalabilità e l'indipendenza dalla topologia, simulando attacchi DoS distribuiti da più host verso un target e traffico legittimo tra altri host.

Il processo di test ha previsto l'avvio del controller, la creazione della topologia desiderata, la generazione di traffico DoS e legittimo, e l'osservazione del comportamento del sistema. Sono stati utilizzati strumenti come hping3, nping e iperf per simulare i diversi tipi di traffico.

Questa metodologia ha permesso di verificare l'efficacia delle logiche di detection e mitigazione, la robustezza del sistema e l'impatto sulle comunicazioni legittime, fornendo una base solida per la valutazione dei risultati.

4.2 Metodologia di Testing

La validazione del sistema NCIS è stata condotta attraverso una serie di simulazioni in ambienti Mininet, utilizzando sia topologie semplici che complesse. Nella topologia semplice, composta da tre host e quattro switch, l'obiettivo era verificare la capacità del controller di distinguere tra traffico DoS e traffico legittimo. Sono stati generati attacchi flood UDP/TCP da un host verso un altro, mentre un terzo host produceva traffico legittimo (ping, iperf).

Nella topologia complessa, con sette host e dieci switch disposti in una struttura mesh/tree-like, si è voluto dimostrare la scalabilità e l'indipendenza dalla topologia del controller. Sono stati simulati attacchi DoS distribuiti da più host verso un target, insieme a traffico legittimo tra altri host.

Il processo di test ha seguito questi passi:

1. Avvio del controller tramite `ryu-manager controller.py`.
2. Avvio della topologia desiderata (`topology.py` per la semplice, `topology_new.py` per la complessa).
3. Generazione di traffico DoS (es. `hping3`, `nping`, flood UDP/TCP) e traffico legittimo (ping, iperf) tra host selezionati.
4. Osservazione del comportamento del sistema, verifica della corretta rilevazione e mitigazione degli attacchi, e analisi dell'impatto sulle comunicazioni legittime.

Questa metodologia ha permesso di testare l'efficacia delle logiche di detection e mitigazione, nonché la robustezza del sistema in scenari realistici e variabili.

Chapter 5

Conclusion

Rispetto alla soluzione presentata lo scorso anno, il progetto NCIS ha subito significativi miglioramenti che ne aumentano la robustezza e l'aderenza agli scenari reali di sicurezza SDN. In primo luogo, il problema dell'over blocking è stato risolto passando da un blocco indiscriminato delle porte degli switch a un controllo granulare sui singoli flussi e indirizzi MAC, con l'adozione di whitelist e blocklist. Questo approccio riduce drasticamente l'impatto sul traffico legittimo e migliora l'esperienza degli utenti. Inoltre, le soglie di rilevamento statiche sono state sostituite da meccanismi adattivi basati su medie mobili e percentili, che permettono al sistema di reagire dinamicamente alle variazioni del traffico di rete, riducendo i falsi positivi. L'architettura è stata resa modulare, separando il monitoraggio, la decisione e l'enforcement in thread distinti: questa scelta incrementa la manutenibilità e facilita l'estensione futura del sistema. Dal punto di vista della detection, il sistema ora è in grado di riconoscere pattern di attacco più complessi, come quelli bursty o distribuiti, grazie all'introduzione di nuove metriche come la varianza del traffico e l'analisi degli intervalli tra i pacchetti. Infine, la gestione delle policy di blocco e sblocco è stata resa più flessibile tramite l'adozione di strategie di backoff esponenziale e sblocco progressivo, evitando penalizzazioni ingiustificate e riducendo il rischio di rientro prematuro degli attaccanti.

In prospettiva futura, il sistema può essere ulteriormente arricchito con algoritmi di machine learning per la detection, integrazione con sistemi di threat intelligence e supporto a protocolli di orchestrazione avanzata. Il lavoro svolto rappresenta una base solida per la ricerca e lo sviluppo di soluzioni di sicurezza SDN sempre più efficaci e adattive.

5.1 Sintesi delle Decisioni Progettuali

Il progetto NCIS è stato sviluppato seguendo un approccio modulare, con la suddivisione delle responsabilità tra i diversi file principali. Il controller (`controller.py`) gestisce l'orchestrazione dei moduli e gli eventi OpenFlow, avviando thread dedicati per monitoraggio, detection, mitigazione e API REST. Il monitor (`monitor.py`) raccoglie periodicamente statistiche granulari dagli switch, fornendo dati aggiornati per la detection. Il detector (`detector.py`) analizza le statistiche tramite plugin, utilizzando soglie adattive per rilevare anomalie e notificando il mitigator. Il mitigator (`mitigator.py`) applica regole di blocco e sblocco granulari, gestendo lo sblocco progressivo e il logging delle decisioni. L'API (`api.py`) espone endpoint REST per la gestione dinamica delle regole.

Le scelte architetturali sono state guidate dalla necessità di garantire modularità,

estendibilità e reattività. L'uso di thread separati assicura monitoraggio e risposta continua, mentre le soglie adattive e il blocco granulare riducono i falsi positivi e l'impatto sul traffico legittimo. La documentazione delle decisioni e la presenza di logging facilitano audit, debugging e future estensioni.

La soluzione è stata progettata per essere facilmente estendibile: nuovi plugin di detection, criteri di mitigazione e endpoint API possono essere aggiunti senza modificare la struttura esistente. Il sistema è pronto per ulteriori sviluppi e integrazioni, mantenendo la flessibilità necessaria per adattarsi a scenari e minacce emergenti.