POLITECNICO DI TORINO
Repository ISTITUZIONALE

A looping process for cyberattack mitigation

Original
A looping process for cyberattack mitigation / Bringhenti, D., Pizzato, F., Sisto, R., Valenza, F.. - ELETTRONICO. - (2024), pp. 276-281. (2024 IEEE International Conference on Cyber Security and Resilience London (UK) 2-4 September 2024) [10.1109/CSR61664.2024.10679501].

Availability:
This version is available at: 11583/2992167 since: 2024-09-25T12:26:44Z

Publisher:
IEEE

Published
DOI:10.1109/CSR61664.2024.10679501

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

19 November 2026

A looping process for cyberattack mitigation

Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, Fulvio Valenza
Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy, Emails: {first.last}@polito.it

***Abstract*—Mitigating cyberattacks in fast times has become a strong requirement for the security management of modern virtual computer networks, where attacks are highly mutable and short-term. Firewalls would still represent an effective defense line, but the traditional manual approaches for their configuration are no longer applicable. Besides, even if automatic approaches for firewall configuration have been recently proposed in literature, they still require excessive interaction with human administrators, thus delaying the attack mitigation. Therefore, this paper proposes a looping autonomous process that mitigates ongoing attacks by reconfiguring distributed firewalls in a provably correct and optimized way. This continuously active process includes a policy extraction engine to extract information from the alerts produced by monitoring agents and to produce security policies whose enforcement would stop the detected attack. An implementation of this multi-step process has been validated in realistic use cases to assess its efficacy and efficiency in stopping cyberattacks.**

***Index Terms*—security automation, attack mitigation, firewall**

I. Introduction

In recent years, dynamism and agility have become key features of next-generation virtual computer networks, thanks to the proliferation of novel paradigms such as network softwarization, cloud, and edge computing. Even though they have been proven significant in enhancing and quickening network management operations such as service function chain provisioning, at the same time, they have led to an increase in network size and complexity. This increase has created new opportunities for attackers to improve their penetration strategies. Recent cyberattacks exploit this higher complexity by rapidly varying how they work to become more difficult to predict. For instance, the burst Distributed Denial of Service (DDoS) attacks that targeted Proton in 2022 were characterized by multiple mutable attack vectors, which could change in the span of a few minutes [1].

Different countermeasures can be used to counter cyberattacks, such as temporarily blocking all communications or raising alerts to administrators. Among them, the most effective response is to promptly reconfigure distributed packet filtering firewalls so as to block the traffic flows identified as malicious during the attack. However, the traditional old-fashioned approaches for firewall configuration cannot cope with the speedy evolutionary trend of modern attacks. In fact, they were completely manual, and therefore they were suitable only for the security management of small-sized networks, where everything was under the direct control of a human administrator. Consequently, the literature started investigating how automation can be leveraged to automate distributed firewall configuration. In particular, several studies adopt policy-based management to automate its configuration, which comprises the definition of the allocation scheme of its multiple instances and the computation of their filtering rules [2]. According to that principle, network administrators can express their desires related to connectivity requirements (i.e., what traffic flows must be blocked or allowed) as policies, which are sentences written with high-level user-friendly languages, and then automatic tools are in charge of refining them into the concrete firewall configuration.

Nevertheless, despite the indisputable progress in firewall configuration automation, the state-of-the-art approaches proposed in literature still require excessive interaction with human administrators [3], thus delaying the actual deployment of the automatically computed configuration, required to stop an ongoing attack. On the one hand, even if the current algorithms employed by Intrusion Detection Systems (IDSs) to identify attacks are getting progressively more intelligent, the administrators must still analyze the IDS log to get the related information, and subsequently they must define new policies manually. On the other hand, the firewall configuration computed by most of the available automatic approaches must be applied manually by the administrators, e.g., they are still in charge of issuing all reconfiguration commands. Both operations are not only slow, but also prone to decision-making errors, which are also due to the variety of networking tools used for intrusion detection of firewall implementations.

In view of these limitations of the state of the art, this paper proposes an autonomous, optimized and provably correct process where the distributed firewall configuration is automatically recomputed after an attack is detected by IDS solutions and applied to the network, without requiring a human user to define personally the new policies for attack management and to issue the low-level configuration commands. This process is designed to be continuously active so as to provide prompt automatic reactions whenever required. A main challenge here was the design of a policy extraction engine capable of extracting crucial information from the alerts produced by monitoring agents and producing a set of security policies that correctly resolve the detected problem. In order to achieve optimization and formal correctness assurance, we decided to use REACT-VEREFOO [4] as firewall reconfiguration module in the implementation of our approach, as it provides the features we were interested in.

The remainder of the paper is structured as follows. Section II discusses related work, underlining its limits with respect to this proposal. Section III illustrates how the envisioned continuous firewall reconfiguration mechanism works. Section

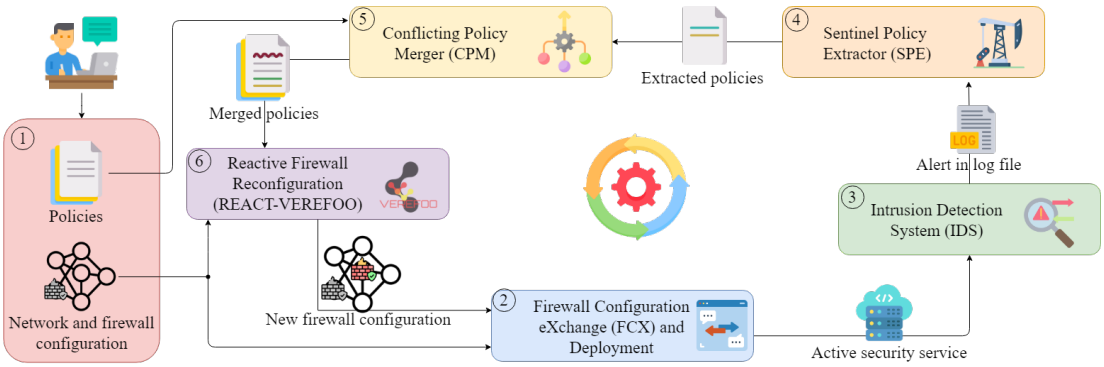


Fig. 1: The proposed approach for autonomous attack mitigation

IV describes its implementation and validation. Section V draws conclusions and discusses future work.

II. Related Work

Mitigating a cyberattack through a distributed firewall is an activity consisting of three main tasks: attack detection, reaction, and firewall reconfiguration. Ideally, all these three operations should be automated, avoiding continuous interventions from human administrators, who could thus limit themselves to supervising the automatic process.

The problems of automating the first and third tasks have already been investigated in the literature, even if in separate ways. For what concerns attack detection, monitoring systems have been recently enhanced with intelligent algorithms such as data-driven techniques [5] and Artificial Intelligence-powered strategies based on Support Vector Machines [6]. Consequently, they are sufficiently autonomous in identifying attacks and raising alerts. For what concerns firewall reconfiguration, the problem of automating it has been addressed by another class of related work [4], [7]–[11]. Among them, the most feature-complete is REACT-VEREFOO [4] because it combines automation, formal verification, and optimization as the main pillars of its algorithm. Specifically, the optimality objective it pursues is to maintain the current firewall configuration as stable as possible during the reconfiguration process, so as to avoid excessive delay in countermeasure provisioning.

However, the second task, i.e., the reaction that should trigger the firewall reconfiguration when prompted by alerts, has not yet been automated. Human administrators are still in charge of analyzing the results produced by the intelligent monitoring systems (e.g., reading the written log files) in order to understand what is actually happening in their managed network. Then, they must manually define the security policies that a tool such as REACT-VEREFOO requires as input to know which traffic flows must be blocked or allowed. Manually performing all these reaction sub-tasks is not only time-consuming, but also prone to errors, e.g., an administrator may incorrectly interpret an alert or define a wrong policy.

Our proposal aims to provide full automation for the reaction task, thus building the missing bridges between the other operations of attack mitigation. Besides, the achievement of this proposal allowed us to formulate a full-fledged process composed of multiple agents, each in charge of automating a specific mitigation sub-task. This process is also flexible enough to be adapted to different network orchestrators, firewall technologies, and monitoring agents.

III. The Proposed Approach

The approach proposed in this paper aims to provide continuous autonomic reconfiguration of distributed packet filtering firewalls in virtual computer networks in an optimized and provably correct way, while avoiding any further human intervention when the process is fully operational. Fig. 1 illustrates the workflow of this approach, whose main phases are described in the following.

1) Input preparation: Initially, the network administrator provides two inputs to the autonomous process, i.e., a set of Network Security Policies (NSPs) and the description of the computer network, including its security configuration required for the satisfaction of all NSPs. Both inputs are formally modeled in our approach, so that other process components can later use them to provide formal assurance about the correctness of the NSP satisfaction.

On the one hand, the first input, i.e., the NSP set, identifies all traffic flows that must be blocked because potentially malicious, or the ones that must reach their destination to provide service connectivity. Each NSP p of this set P_c, named *current NSP set* as it includes the NSPs that are currently satisfied in the network, is formally modeled as a tuple (a, C), where $p.a$ is the action that must be applied on packets matching with the condition predicate $p.C$. This paper uses the "." symbol to refer to tuple elements. In greater detail, $p.C$ is a conjunction of five sub-predicates, each expressing a condition on a specific field of the five composing the IP 5-tuple. For conciseness, this conjunction is symbolized as $p.C = (IPSrc, IPDst, pSrc, pDst, tProto)$. An NSP is defined isolation policy if the action $p.a$ is "deny", reachability policy if instead the action $p.a$ is "allow".

On the other hand, the second input, i.e., the one related to the administrated network, provides all information about

the current network status, including its topological structure and the configuration of all network middleboxes or firewalls. The network is thus modeled as a directed graph $G = (N, L)$, where N is the node set, L is the link set. Specifically, N includes different node subsets depending on their functionality: $E \subseteq N$ is the set of endpoints (e.g., web clients and servers), $M \subseteq N$ is the set of network middleboxes (e.g., NATs and load balancers), $F \subseteq N$ is the set of already present firewalls. Each $f \in F$ is also characterized by a default action ("deny" or "allow") and a set of filtering rules, which may block or permit specific kinds of traffic depending on the 5-tuple field values, as these firewalls are packet filters. Additionally, N includes a particular subset A, including the so-called Allocation Places (APs). These APs represent all the logical positions in the virtual network where new firewalls may be allocated in the future, e.g., if there is the need to have a new firewall after an attack detection. Each flow $w \in W$, where W is the set of all flows that may cross the network G, is modeled as a list of alternating nodes and packet classes $[n_s, t_{sa}, n_a, t_{ab}, n_b, ..., n_z, t_{zd}, n_d]$, where each node n_i in the list represents a node crossed by the flow, whereas the traffic t_{ij} is again a conjunction of five sub-predicates, one for each IP 5-tuple field, and it represents the class of packets transmitted from node n_i to node n_j. Each t_{ij} predicate has the same formal model as the $p.C$ predicate of the NSPs.

The formal models representing these two inputs are defined with a medium-level language independent of low-level network settings. For example, this language may be based on XML or JSON, thus abstracting from the settings for specific firewall implementations.

Besides, the definition of the firewall configuration required to satisfy the current NSPs for this network may be defined manually by the administrator, or it may be computed automatically with state-of-the-art tools such as VEREFOO [2]. In the former case, the administrator must perform more manual operations. In the latter, he simply feeds the automatic tool with the information requested for firewall configuration. Anyhow, this task represents the only human interaction point with our proposed autonomic methodology.

2) Firewall configuration translation and deployment: Supposing that all the elements of the network N are already active Virtual Network Functions (VNFs) in the network (e.g., Virtual Machines orchestrated by Open Source MANO or containers managed by Docker Compose), the missing element to enforce the requested NSPs P_c consists in provisioning virtual functions working as firewalls and deploying their configuration. Nevertheless, as discussed before, their configuration is defined in an implementation-agnostic way, so it cannot be directly applied to the instantiated VNFs. Therefore, at this stage, our approach includes a translating agent named Firewall Configuration eXchange (FCX), which can convert XML/JSON files into the specific commands of different firewall implementations. This agent supports the most commonly used softwarized firewall technologies: iptables, ipfirewall, eBPF firewall, Open vSwitch. However, it has been designed to be flexible enough to be extended easily to other technologies, thus providing forward compatibility.

After this translation, the configuration settings are passed on to the network orchestrator, which can thus apply them to the firewalling VNFs. From this moment, all the current NSPs are concretely enforced in the network.

3) Attack detection: Some VNFs deployed in the virtual network are monitoring agents, i.e., IDSs inspecting traffic flowing through their monitored interfaces to search for potential attacks. The IDS detection engine evaluates the received packets against a set of rules expressing the conditions under which the IDS should assume that a specific attack is occurring. When the conditions specified in a rule are met, the IDS triggers an alert by creating a new entry in its log file, describing all the information deduced by the IDS through its monitoring activity. Clearly, IDSs can produce false positives. However, this is intrinsic to their behavior, and the administrator can remedy the problem by monitoring the process, detecting the false positives, and applying appropriate corrections. Besides, as already discussed in Section II, the literature is rich in proposals about intelligent strategies for attack detection. So, our study is not interested in researching a new one, but our approach can support any IDS technology as long as the IDS produces a log file that the next steps of our methodology can later use.

4) NSP extraction: A core component of the autonomous process that we propose is a vigilant agent named Sentinel Policy Extractor (SPE). The SPE is granted read permission to the log files written by the IDSs installed in the network, so that it can periodically check them to search for the inclusion of new entries related to attack detection. Whenever such an entry is detected, the SPE extracts crucial information and produces a set of NSPs that should correctly stop the attack.

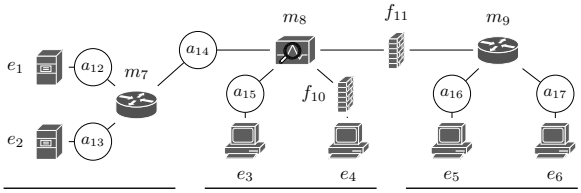
The algorithm that we defined for this policy extraction engine works as follows. First, whenever a log entry about attack detection is detected, the SPE applies an abstract model of the alarm to that implementation-specific entry, so as to retrieve the information representing the formal model of the traffic (i.e., packet class) t_m that causes the alarm. In particular, as $t_m = (\mathit{IPSrc}, \mathit{IPDst}, \mathit{pSrc}, \mathit{pDst}, \mathit{tProto})$, the required information is composed of the values for the IP 5-tuple fields. If no information about any of these fields is missing in the log entry, this means that any possible value is acceptable for it, and this is represented by the wildcard * symbol for that field sub-predicate of the traffic model. Second, starting from t_m, the SPE identifies all traffic flows where this packet class appears, creating a subset W_m of potentially malicious flows. In fact, it is possible that the malicious packets can come from multiple sources. The set of possible flows crossing the network W was part of the input of the process, so the SPE can derive W_m as follows: $\forall w \in W.\ (\exists t_{ij} \in w.\ t_{ij} = t_m) \implies w \in W_m$. Third, for each flow $w \in W_m$ such that $w = [n_s, t_{sa}, n_a, ..., t_m, ..., n_z, t_{zd}, n_d]$, the SPE extracts a policy $p_w = (t_{sa}.\mathit{IPSrc}, t_{zd}.\mathit{IPDst}, t_{sa}.\mathit{pSrc}, t_{zd}.\mathit{pDst}, t_{sa}.\mathit{tProto})$. The sub-predicates of p_w related to the source IP address and port are taken from the initial flow packet class t_{sa}, whereas the sub-predicates

related to the destination fields are taken from the final flow packet class t_{zd}. The reason why those predicates are not directly mutated from t_d is that some packet fields may have been modified by intermediate nodes such as NATs, while the conditions of an NSP should be defined over the end-to-end communication against which protection is required. The whole set of extracted NSPs is denoted as P_e.

5) NSP merge: After extracting P_e, these NSPs should be merged with the NSPs in P_c, which are currently satisfied by the existing firewall configuration. However, this merge must be performed taking into account that some extracted NSPs may conflict with the current ones. For example, a traffic flow that had to reach its destination according to a reachability policy of P_c must now be blocked because of a newly extracted isolation policy of P_e.

This problem is addressed by another core architectural component of our process, named Conflicting Policy Merger (CPM), whose objective is to create a merged NSP set P_m. Clearly, the CPM directly includes all the extracted policies into P_m because their enforcement is strictly required to stop the attack. Then, for each $p \in P_c$, the CPM checks if it conflicts with any $p' \in P_e$ to decide if and how to include it. In particular, two NSPs p and p' are considered conflicting if they have different actions and (at least partially) overlapping conditions, i.e., if $p.a = p'.a$ and $p.C \wedge p'.C$. Given this premise, if $p \in P_c$ does not conflict with any extracted NSP, it is directly included by the CPM into the merged set P_m. Instead, if it conflicts with some $p' \in P_e$, the CPM must modify its predicate $p.C$ so that it does not include the overlapped part anymore, and then it can put the modified NSP into the merged set. For example, if a reachability policy was previously applied to the TCP traffic targeting all ports in the range [50200, 50300] while now an extracted isolation policy imposes that TCP traffic to 50300 must be blocked because exploited by an attack, then the original reachability policy must be modified so that the condition on the destination port is defined on the restricted interval [50200, 50209].

6) Firewall reconfiguration computation: After the computation of the merged NSP set P_m, the new firewall configuration (composed of its new allocation scheme and distributed rule set) is automatically computed. For this operation, our process uses the most feature-complete algorithm in literature, REACT-VEREFOO [4]. This algorithm formulates the firewall reconfiguration problem as a Maximum Satisfiability Modulo Theories (MaxSMT) through constraint programming. This formulation provides automation, formal verification, and optimization. First, the problem is solved automatically by executing an SMT solver. Second, the output is proved correct a-priori, as long as all inputs are formally modeled in a way that captures all the information that may impact the solution correctness. Third, the presence of some relaxable clauses, named soft constraints, enables the achievement of optimization goals. Here, the objective pursued by REACT-VEREFOO is to modify the firewall configuration while keeping the current one as much as possible (e.g., it is preferable to maintain all firewalls in their current position and modify their rule sets, as otherwise a larger time would be needed to deploy a new VNF to stop the attack).



Endpoint	IP address
e_1	120.48.0.2
e_2	120.48.0.3

Endpoint	IP address
e_3	42.105.2.2
e_4	42.105.2.3

Endpoint	IP address
e_5	136.10.0.2
e_6	136.10.0.3

Fig. 2: Current firewall allocation scheme

Firewall f_{10}

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	42.105.2.3	120.48.0.∗	∗	30120	UDP
1	Allow	120.48.0.∗	42.105.2.3	30120	∗	UDP
3	Allow	42.105.2.3	42.105.2.2	∗	∗	∗
D	Deny	∗.∗.∗.∗	∗.∗.∗.∗	∗	∗	∗

Firewall f_{11}

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	136.10.0.∗	120.48.0.2	∗	∗	TCP
2	Allow	136.10.0.3	42.105.2.2	∗	32456	TCP
3	Allow	42.105.2.2	136.10.0.3	32456	∗	TCP
D	Deny	∗.∗.∗.∗	∗.∗.∗.∗	∗	∗	∗

TABLE I: Current firewall filtering rules

REACT-VEREFOO is thus simply fed with the description of the current network and firewall configuration N and with the merged NSP set P_m. Through its internal algorithm, it provides a description of the new firewall configuration, expressed in implementation-independent medium-level language.

This new configuration is parsed by the FCX agent. Then, the orchestrator deploys new firewalling VNFs if required, and it applies the low-level firewall configuration settings to the corresponding VNFs. The attack is thus blocked because all new policies are enforced successfully, and the administrator also has formal proof of this achievement thanks to the usage of REACT-VEREFOO. At this stage, the process starts to loop because the monitoring agents continue to detect other attacks, and whenever an alert is raised, the SPE is ready to extract a new policy and trigger a new reconfiguration. All operations continue to work without the active intervention of the human administrator, who can simply supervise and ensure that everything continues to behave as expected.

IV. Implementation and Validation

The implementation of this process required defining the policy language, developing multiple agents, extending existing tools, and providing the necessary communication interfaces among them. The medium-level language chosen to represent all policies and firewall configuration is the Medium Security Policy Language (MSPL), a well-known language based on XML that has been used and validated in multiple EU-funded projects such as SECURED and ANASTASIA. The translator agent FCX has been developed in Java to exploit the JAXB library, which automates the mapping between XML

Action	IPSrc	IPDst	pSrc	pDst	tProto
Allow	136.10.0.∗	120.48.0.2	∗	∗	TCP
Allow	42.105.2.3	120.48.0.2	∗	30120	UDP
Allow	42.105.2.3	120.48.0.3	∗	30120	UDP
Allow	120.48.0.2	42.105.2.3	30120	∗	UDP
Allow	120.48.0.3	42.105.2.3	30120	∗	UDP
Allow	42.105.2.3	42.105.2.2	∗	∗	∗
Allow	136.10.0.3	42.105.2.2	∗	32456	TCP
Allow	42.105.2.2	136.10.0.3	32456	∗	TCP
Allow	136.10.0.∗	136.10.0.∗	∗	∗	∗
Deny	42.105.2.2	42.105.2.3	∗	∗	∗
Deny	42.105.2.3	136.10.0.∗	∗	∗	∗
Deny	136.10.0.∗	42.105.2.3	∗	∗	∗
Deny	136.10.0.∗	120.48.0.3	∗	∗	∗
Deny	120.48.0.3	136.10.0.∗	∗	∗	∗

TABLE II: Current network security policies

documents and Java objects, and it can produce low-level configuration settings for iptables, ipfirewall, Open vSwitch, and eBPF firewall. The sentinel agent SPE has been developed in python3, and it can parse log files produced by Snort3 and OSSEC3.7, two extensively adopted IDS implementations. Similarly, the merging agent CPM has been written in python3. These agents have interfaces by which they can exchange data. Finally, the REACT-VEREFOO implementation has been extended with REST APIs that can be used by other elements of our architecture (e.g., the CPM agent to trigger a reconfiguration), and the orchestrator employed for VNF management is Docker Compose. However, our process is general enough to work with other technologies (e.g., Suricata as IDS or Open Source MANO as orchestrator). It would be enough to modify or extend the current agent prototypes to support them.

This implementation has been validated in two realistic use cases within the scope of the SERICS project. The first use case is about the mitigation of a DoS attack exploiting a backdoor on TCP port 7597, which may have been opened by worms such as QAZ. Instead, the second one is related to a TCP SYN port scan. For space limitation, here we report only the description of the latter. This use case is based on the virtual network illustrated in Fig. 2, where each graph node is containerized as a Docker. The elements with the a notation are the APs where future firewalls may be included to stop attacks. Instead, the elements with the f notation are already allocated firewalls, implemented with iptables. Their configuration rules, expressed in TABLE I, allow enforcing all currently desired NSPs, listed in TABLE II. It is worth noting that there is no one-to-one correspondence between firewall rules and NSPs for multiple reasons. On the one hand, in this use case, firewalls employ default actions to provide whitelisting configuration. On the other hand, the administrator uses aggregated rules to satisfy multiple NSPs when possible.

In this scenario, the objective is to detect and stop a specific class of attacks, the TCP SYN port scan, frequently used to determine the state of TCP ports without establishing a full connection, so as to understand if there are open ports exploitable to carry out an attack to the victim node. For this purpose, the middlebox m_8 is made to work as a monitoring agent, as OSSEC3.7 is installed in the corresponding container, with the configuration listed in Listing 1.

```
<rule id="100009" level="1">
  <options>no_log</options>
  <decoded_as>iptables</decoded_as>
  <description>TCP SYN request detected</description>
</rule>
<rule id="100010" level="10" frequency="20" timeframe="60">
  <if_matched_sid>100009</if_matched_sid>
  <decoded_as>iptables</decoded_as>
  <description>TCP SYN port scan detected</description>
  <same_source_ip />
</rule>
```

Listing 1: OSSEC configuration

Rule 100009 matches any TCP SYN request without logging it. This rule assists rule 100010, which activates when at least 20 TCP SYN requests from the same source IP are identified within 60 seconds, signaling a potential port scan.

The attack simulation starts by accessing the containerized en point e_5 via a shell:

```
sudo docker exec -it e5 /bin/sh
```

The port scan is simulated through a simple command employing `netcat`. This method is preferred over just using `nmap` to minimize the need for extra software installations on endpoints. The specific command used for the attack simulation is the following:

```
for p in $(seq 1 25); do nc 120.48.0.2 $p; done
```

This command sends 25 TCP SYN requests to e_1, triggering the rule in OSSEC and resulting in the following alert, represented by the log entry listed in Listing 2.

```
{
  "rule": {
    "comment": "TCP SYN port scan detected",
    "sidid": 100010,
    "frequency": 20,
  },
  "protocol": "TCP",
  "srcip": "136.10.0.2",
  "dstip": "120.48.0.2",
  "agent_name": "m8",
  "timestamp": "2024 May 13 10:35:57",
  "logfile": "/var/log/ulog/syslogemu.log"
}
```

Listing 2: Log entry notifying the attack detection

As soon as the SPE agent detects that a new log entry has been written by the IDS, it automatically parses it and extracts an isolation policy written with the MSPL language, formatted as the XML object reported in Listing 3.

```
<PropertyDefinition>
   <Property graph="0" name="IsolationProperty"
     src="136.10.0.2" dst="120.48.0.2" lv4proto="TCP"/>
</PropertyDefinition>
```

Listing 3: Extracted NSP

Next, the CPM agent merges the extracted NSP with the previously enforced NSPs. The only conflict occurs with the first NSP of the ones listed in TABLE II. In fact, that reachability policy requested that the TCP traffic from all endpoints with IP addresses in the 136.10.0.0/24 range must be able to contact the endpoint with IP address 120.48.0.2. However, this is not acceptable anymore, because the TCP traffic from 136.10.0.2 must now be stopped. Therefore, the

Firewall f_{11}

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	136.10.0.3	120.48.0.2	*	*	TCP
2	Allow	136.10.0.3	42.105.2.2	*	32456	TCP
3	Allow	42.105.2.2	136.10.0.3	32456	*	TCP
D	Deny	*.*.*.*	*.*.*.*	*	*	*

TABLE III: Updated firewall filtering rules

CPM modifies the first NSP of TABLE II so that the only acceptable source IP address is 136.10.0.3, and adds the newly extracted isolation policy within this list.

REACT-VEREFOO is fed with this merged NSP set to identify the optimal firewall reconfiguration solution that could stop the attack. In fact, multiple correct solutions exist. For example, a human administrator may decide to directly instantiate a new firewalling container in a_{12}, a_{14}, or a_{16} to block the malicious traffic selectively. However, the provisioning time of a new Docker is not negligible, especially because it is mainly impacted by the image import, which may last tens of seconds according to [12]. The provisioning time would be even worse if the administrator used Open Source MANO instead of the Docker Compose orchestrator used for our test bed. In fact, according to [13], the Deployment Process Delay (DPD) introduced by Open Source MANO to deploy and instantiate a VNF within an already booted VM and set up an operational network service is 134ms. Instead, as also proved by its creators in [4], REACT-VEREFOO optimally produces the reconfiguration solution where no new firewalls are added, but the filtering rules of the firewall f_{11} are simply modified. The new rules are expressed by REACT-VEREFOO in MSPL with a similar format as Listing 3, but for space limitation are reported here in compacted form in TABLE III.

Then, the FCX agent translates the MSPL configuration to iptables commands, producing the script of Listing 4.

```
#!/bin/sh
cmd="sudo iptables"
${cmd} -F
${cmd} -P INPUT DROP
${cmd} -P FORWARD DROP
${cmd} -P OUTPUT DROP
${cmd} -A FORWARD -p tcp -s 136.10.0.3/32 -d
    120.48.0.2/32 --dport ACCEPT
${cmd} -A FORWARD -p tcp -s 136.10.0.3/32 -d
    42.105.2.2/32 --dport 32456 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 42.105.2.2/32 -d
    136.10.0.3/32 --sport 32456 -j ACCEPT
```

Listing 4: Translated iptables configuration

As soon as the configuration of f_{11} is concretely updated, the attack is successfully stopped. This result has been experimentally confirmed by reapplying the command to perform a port scan from e_5 to e_1, which this time turns unsuccessful.

The implementation of this process is also efficient in terms of performance. The FCX, SPE, and CPM agents require less than a second each to execute the tasks they are in charge of. The most time-consuming module is REACT-VEREFOO. However, as experimentally shown in [4], it takes around 4 seconds to recompute a firewall configuration in front of a change of 10-20% of 30 original NSPs. Such execution time is quite lower than the time requested by orchestrators to instantiate a new VNF, so it neither is a bottleneck nor a significant delay contribution to attack mitigation.

V. Conclusion and Future Work

This paper proposed a full-fledged looping process for attack mitigation, based on an optimized and formally correct firewall reconfiguration. This process integrates multiple agents to avoid external human interventions, and provides an intelligent engine to derive security policies from alerts raised by monitoring systems. The validation of our prototype implementation of this process in realistic use cases showed its efficacy and efficiency in stopping attacks.

Future work envisions extending this process to the mitigation of attacks for which the reconfiguration of other firewall types may be required, e.g., web-application firewalls. We will then investigate possible limitations of the proposed approach, such as DoS attacks that may be carried out by continuously triggering firewall reconfiguration. Extensive validation of the process will also be continued in the SERICS project.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the EU - NextGenerationEU.

References

[1] Proton, "A brief update regarding ongoing DDoS incidents," 2022, Available: https://proton.me/blog/a-brief-update-regarding-ongoing-ddos-incidents, Visited: 2024-05-07.

[2] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1559–1576, 2023.

[3] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 57:1–57:37, 2024.

[4] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration," in *Proc. of IEEE/IFIP Network Operations and Management Symposium, Seoul, South Korea*, 2024.

[5] D. Chou and M. Jiang, "A survey on data-driven network intrusion detection," *ACM Comput. Surv.*, vol. 54, no. 9, pp. 182:1–182:36, 2022.

[6] M. Mohammadi, T. A. Rashid, S. H. T. Karim, A. H. M. Aldalwie, Q. T. Tho, M. Bidaki, A. M. Rahmani, and M. Hosseinzadeh, "A comprehensive survey and taxonomy of the svm-based intrusion detection systems," *J. Netw. Comput. Appl.*, vol. 178, p. 102983, 2021.

[7] F. Chen, A. X. Liu, J. Hwang, and T. Xie, "First step towards automatic correction of firewall policy faults," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 2, pp. 1–24, 2012.

[8] N. B. Youssef and A. Bouhoula, "A fully automatic approach for fixing firewall misconfigurations," in *Proc. of 11th IEEE Int. Conf. on Computer and Information Technology*, 2011, pp. 461–466.

[9] K. Adi, L. Hamza, and L. Pene, "Automatic security policy enforcement in computer systems," *Comput. Secur.*, vol. 73, pp. 156–171, 2018.

[10] W. T. Hallahan, E. Zhai, and R. Piskac, "Automated repair by example for firewalls," *Formal Methods Syst. Des.*, vol. 56, no. 1, pp. 127–153, 2020.

[11] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A comprehensive approach to the automatic refinement and verification of access control policies," *Comput. Secur.*, vol. 80, pp. 186–199, 2019.

[12] B. Xavier, T. Ferreto, and L. C. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *Proc. of IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, Cartagena, Colombia, 2016*, 2016, pp. 277–280.

[13] G. M. Yilma, F. Z. Yousaf, V. Sciancalepore, and X. P. Costa, "Benchmarking open source NFV MANO systems: OSM and ONAP," *Comput. Commun.*, vol. 161, pp. 86–98, 2020.