

Towards Filtering and Alerting Rule Rewriting on Single-Component Policies

J. G. Alfaro^{1,2}, Frédéric Cuppens¹, and Nora Cuppens-Boulahia¹

¹ GET/ENST-Bretagne, 35576 Cesson Sévigné - France
{frederic.cuppens,nora.cuppens}@enst-bretagne.fr

² dEIC/UAB, Edifici Q, 08193, Bellaterra, Barcelona - Spain
Joaquin.Garcia-Alfaro@deic.uab.es

Abstract: The use of *firewalls* and *network intrusion detection systems* (NIDSs) is the dominant method to survey and guarantee the security policy in current corporate networks. On the one hand, firewalls are traditional security components which provide means to filter traffic within corporate networks, as well as to police the incoming and outgoing interaction with the Internet. On the other hand, NIDSs are complementary security components used to enhance the visibility level of the network, pointing to malicious or anomalous traffic. To properly configure both firewalls and NIDSs, it is necessary the use of a set of configuration rules, i.e., a set of filtering or alerting rules. Nevertheless, the existence of anomalies within the set of configuration rules of both firewalls and NIDSs is very likely to degrade the network security policy. The discovering and removal of these anomalies is a serious and complex problem to solve. In this paper, we present a set of mechanisms for such a management.

Keywords: Network Security, Firewalls, NIDSs, Policy Anomalies

1 Introduction

Many companies and organizations use *firewalls* to police their incoming and outgoing flow of traffic between different zones of the network, as well as *network intrusion detection systems* to monitor and survey such a traffic. A firewall is a network security component, with several interfaces associated with the different zones of the network. The company may partition, for instance, its network into three different zones: a *demilitarized zone* (or DMZ), a private network and a zone for security administration. This way, one may use a single firewall setup, with three interfaces associated with these three zones, to police the protection of each zone³. Network intrusion detection systems (NIDSs for short), on the other hand,

³ Firewalls also implement other functionalities, such as Proxying and Network Address Transfer (NAT), but it is not the purpose of this paper to cover these functionalities.

are complementary network security components which are in charge of detecting malicious or anomalous activity in the network traffic, such as *denial of service* (DoS) attacks or intrusion attempts. NIDSs can employ different families of detection methods, being *anomaly detection* and *misuse detection* two of the most frequently used methods. We refer to [9] for a good survey on the field.

In order to apply a filtering policy, it is necessary to configure the firewall with a set of filtering rules. Similarly, and in order to apply an alerting policy, it is also necessary to configure the NIDS with a set of alerting rules (i.e., *detection signatures* when using misuse detection methods). Both filtering and alerting rules are specific cases of a more general configuration rule, which typically defines a *decision* (such as *filter*, *alert*, *pass*, etc.) that applies over a set of *condition* attributes, such as *protocol*, *source*, *destination*, *classification*, etc.

For our work, we define a general configuration rule as follows:

$$R_i : \{condition_i\} \rightarrow decision_i \quad (1)$$

where i is the relative position of the rule within the set of rules, $\{condition_i\}$ is the conjunctive set of condition attributes such that $\{condition_i\}$ equals $C_1 \wedge C_2 \wedge \dots \wedge C_p$ – being p the number of condition attributes of the given rule – and *decision* is a boolean value in $\{true, false\}$.

Let us notice that the decision of a filtering rule will be positive (*true*) whether it applies to a specific value related to *deny* (or *filter*) the traffic it matches, and will be negative (*false*) whether it applies to a specific value related to *accept* (or *pass*) the traffic it matches. Similarly, the decision of an alerting rule will be positive (*true*) whether it applies to a specific value related to *warn* (or *alert*) about the traffic it matches, and will be negative (*false*) whether it applies to a specific value related to *ignore* the traffic it matches.

In the configuration policy of a component, conflicts due to rule overlaps, i.e., the same traffic matching more than one rule, can occur. To solve these conflicts, most components implement a *first matching* strategy through the ordering of rules. This way, each packet processed by the component is mapped to the decision of the rule with highest priority. This strategy introduces, however, new configuration errors, often referred in the literature as *policy anomalies*.

In [5], we presented an audit process to manage firewall policy anomalies, in order to detect and remove anomalies within the set of rules of a given firewall. This audit process is based on the existence of relationships between the condition attributes of the filtering rules, such as coincidence, disjunction, and inclusion, and proposes a transformation process which derives from an initial set of rules – with potential policy anomalies – to an equivalent one which is completely free of such anomalies.

In this paper, we extend our proposal of detecting and removing firewall policy anomalies [5], to a more complete setup where both firewalls and NIDSs are in charge of the network security policy. Hence, assuming that the role of both prevention and detection of network attacks is assigned to these two components, our objective is to completely correct the anomalies within their configuration.

We also extend in this paper the set of anomalies studied in [5] which, in turn, are not reported, as defined in this paper, in none of the studied related work. For such a purpose, we also introduce in this paper the use of a model to specify some properties of the network, e.g., vulnerabilities, as well as to determine whether the network traffic that matches a given configuration rule, may or may not cross the component configured by such a rule.

The advantages of our proposal are threefold. First, when performing our proposed discovery and removal of anomalies, and after rewriting the rules, one can verify that the resulting configuration of each component in the network is free of misconfiguration. Each anomalous rule will be reported to the administration console. This way, the security officer in charge of the network can check the network policy, in order to verify the correctness of the whole process, and perform the proper policy modifications to avoid such anomalies.

Second, the resulting rules are totally disjoint, i.e., the ordering of rules is no longer relevant. Hence, one can perform a second transformation in a positive or negative manner, generating a configuration that only contains positive rules if the component default policy is negative, and negative rules if the default policy is positive.

Third, the set of configuration rules enhanced through our algorithms may significantly help to reduce the number of false positive events warned by NIDSs (i.e., alerts that the NIDS reports when it is not supposed to) since we best fit the number and type of alerting rules to the network properties.

The rest of this paper is organized as follows. Section 2 starts by introducing a network model that is further used in Section 3 when presenting our set of algorithms. Section 4 overviews the performance of our proposed algorithms, and Section 5 introduces an analysis of some related work. Finally Section 6 closes the paper with some conclusions.

2 Network Model

The purpose of our network model is to determine whether the traffic that matches a given configuration rule R_i may or may not cross the component configured by such a rule. It is defined as follows. First, and concerning the traffic flowing from two different zones of the network, we may determine the set of components that are crossed by this flow. Regarding the scenario shown in Figure 1, for example, the set

of components crossed by the network traffic flowing from zone *external network* to zone *private₃* equals $[C_1, C_2, C_4]$, and the set of components crossed by the network traffic flowing from zone *private₃* to zone *private₂* equals $[C_4, C_2, C_3]$.

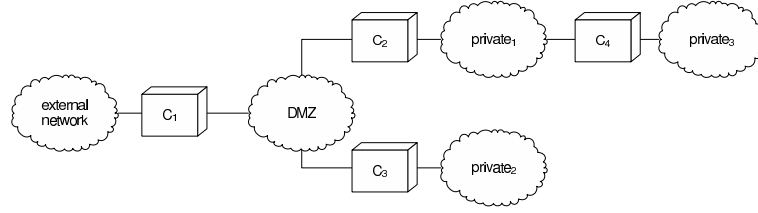


Fig. 1. Simple distributed policy setup.

Let C be a set of components and let Z be a set of zones. We assume that each pair of zones in Z are mutually disjoint, i.e., if $z_i \in Z$ and $z_j \in Z$ then $z_i \cap z_j = \emptyset$. We then define the predicate $connected(c_1, c_2)$ as a symmetric and anti-reflexive function which becomes *true* whether there exists, at least, one interface connecting component c_1 to component c_2 . On the other hand, we define the predicate $adjacent(c, z)$ as a relation between components and zones which becomes *true* whether the zone z is interfaced to component c . Referring to Figure 1, we can verify that predicates $connected(C_1, C_2)$ and $connected(C_1, C_3)$, as well as $adjacent(C_1, DMZ)$, $adjacent(C_2, private_1)$, $adjacent(C_3, DMZ)$, and so on, become *true*.

We then define the set of paths, P , as follows. If $c \in C$ then $[c] \in P$ is an atomic path. Similarly, if $[p.c_1] \in P$ (be “.” a concatenation functor) and $c_2 \in C$, such that $c_2 \notin p$ and $connected(c_1, c_2)$, then $[p.c_1.c_2] \in P$. This way, we can notice that, concerning Figure 1, $[C_1, C_2, C_4] \in P$ and $[C_1, C_3] \in P$.

Let us now define a set of functions related with the order between paths. We first define functions *first*, *last*, and the order functor between paths. We first define function *first* from P in C such that if p is a path, then $first(p)$ corresponds to the first component in the path. Conversely, we define function *last* from P in C such that if p is a path, then $last(p)$ corresponds to the last component in the path. We then define the order functor between paths as $p_1 \leq p_2$, such that path p_1 is shorter than p_2 , and where all the components within p_1 are also within p_2 .

Two additional functions are *route* and *minimal_route*. We first define function *route* from Z to Z , i.e., $\{route(z_1, z_2) : Z \times Z \text{ in } 2^P\}$, such that $p \in route(z_1, z_2)$ iff the path p connects zone z_1 to zone z_2 . Formally, we define $p \in route(z_1, z_2)$ iff $adjacent(first(p), z_1)$ and $adjacent(last(p), z_2)$. Similarly, we then define *minimal_route* from Z to Z , i.e., $\{minimal_route(z_1, z_2) : Z \times Z \text{ in } 2^P\}$, such that $p \in minimal_route(z_1, z_2)$ iff the following conditions hold: (1) $p \in route(z_1, z_2)$; (2) There does not exist $p' \in route(z_1, z_2)$ such that $p' < p$.

Regarding Figure 1, we can verify that the *minimal_route* from zone *private₃* to zone *private₂* equals $[C_4, C_2, C_3]$, i.e., $minimal_route(private_3, private_2) = \{[C_4, C_2, C_3]\}$.

Let us finally conclude by defining the predicate *affects*(Z, A_c) as a boolean expression which becomes *true* whether there is, at least, an element $z \in Z$ such that the configuration of z is vulnerable to the attack category $A_c \in V$, where V is a vulnerability set built from a vulnerability database, such as CVE[8] or OSVDB[10].

3 Our Proposal

In this section we present our set of audit algorithms, whose main objective is the complete discovering and removal of policy anomalies that could exist in a single component policy, i.e., to discover and warn the security officer about potential anomalies within the configuration rules of a given component. Let us start by classifying the complete set of anomalies of our proposal.

3.1 Classifying the Anomalies

We classify in this section the complete set of anomalies that can occur within a single component configuration. An example for each anomaly will be illustrated through the sample scenarios shown in Figure 2.

Shadowing A configuration rule R_i is shadowed in a set of configuration rules R whether such a rule never applies because all the packets that R_i may match, are previously matched by another rule, or combination of rules, with higher priority in order. Regarding Figure 2, rule $C_1\{R_6\}$ is shadowed by the overlapping of rules $C_1\{R_3\}$ and $C_1\{R_5\}$.

Redundancy A configuration rule R_i is redundant in a set of rules R whether the rule is not shadowed by any other rule or set of rules and, when removing R_i from R , the security policy does not change. For instance, referring to Figure 2, rule $C_1\{R_4\}$ is redundant, since the overlapping between rules $C_1\{R_3\}$ and $C_1\{R_5\}$ is equivalent to the police of rule $C_1\{R_4\}$.

Irrelevance A configuration rule R_i is irrelevant in a set of configuration rules R if one of the following conditions holds:

- (1) Both source and destination address are within the same zone, and its decision is *false*. For instance, rule $C_1\{R_1\}$ is irrelevant since the source of this address, *external network*, as well as its destination, is the same.
- (2) The component is not within the minimal route that connects the source zone, concerning the irrelevant rule which causes the anomaly, to the destination zone.

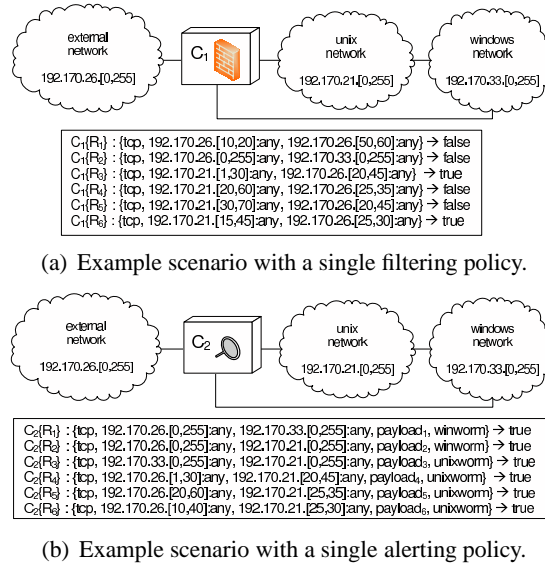


Fig. 2. Example filtering and alerting policies.

Hence, the rule is irrelevant since it matches traffic which does not flow through this component. Rule $C_2\{R_3\}$, for example, is irrelevant since component C_2 is not in the path which corresponds to the minimal route between the source zone *windows network* to the destination zone *unix network*.

(3) At least one of the condition attributes in R_i is related with a classification of attack A_c which does not affect the destination zone of such a rule, i.e., the predicate $\text{affects}(z_d, A_c)$ becomes *false*. Regarding Figure 2, we can see that rule $C_2\{R_2\}$ is irrelevant since the nodes in the destination zone *unix network* are not affected by vulnerabilities classified as *winworm*.

3.2 Proposed Algorithms

Our proposed audit process is a way to alert the security officer in charge of the network about these configuration errors, as well as to remove all the useless rules in the initial component configuration. The data to be used for the detection process is the following. A set of rules R as a list of initial size n , where n equals $\text{count}(R)$, and where each element is an associative array with the strings *condition*, *decision*, *shadowing*, *redundancy*, and *irrelevance* as keys to access each necessary value.

For reasons of clarity, we assume one can access a linked-list through the operator R_i , where i is the relative position regarding the initial list size – $\text{count}(R)$. We

also assume one can add new values to the list as any other normal variable does ($element \leftarrow value$), as well as to remove elements through the addition of an empty set ($element \leftarrow \emptyset$). The internal order of elements from the linked-list R keeps with the relative ordering of rules.

Each element $R_i[condition]$ is a boolean expression over p possible attributes. To simplify, we only consider as attributes the following ones: $szone$ (source zone), $dzone$ (destination zone), $sport$ (source port), $dport$ (destination port), $protocol$, and $attack_class$ – or A_c for short – which will be empty whether the component is a firewall. In turn, each element $R_i[decision]$ is a boolean variable whose values are in $\{true, false\}$. Finally, elements $R_i[shadowing]$, $R_i[redundancy]$, and $R_i[irrelevance]$ are boolean variables in $\{true, false\}$ – which will be initialized to $false$ by default.

We split the whole process in four different algorithms. The first algorithm (cf. Algorithm 1) is an auxiliary function whose input is two rules, A and B . Once executed, this auxiliary function returns a further rule, C , whose set of condition attributes is the exclusion of the set of conditions from A over B . In order to simplify the representation of this algorithm, we use the notation A_i as an abbreviation of the variable $A[condition][i]$, and the notation B_i as an abbreviation of the variable $B[condition][i]$ – where i in $[1, p]$.

The second algorithm is a boolean function in $\{true, false\}$ which applies the necessary verifications to decide whether a rule r is irrelevant for the configuration of a component c . To properly execute this algorithm, let us define Z as the set of zones, $source(r)$ as a function in Z such that $source(r) = szone$, and $dest(r)$ as a function in Z such that $dest(r) = dzone$.

The third algorithm is a boolean function in $\{true, false\}$ which, in turn, applies the transformation *exclusion* (cf. Algorithm 1) over a set of configuration rules to check whether the rule obtained as a parameter is potentially redundant.

The last algorithm (i.e., Algorithm 4) performs the whole process of detecting and removing the complete set of anomalies. This process is split in three different phases. During the first phase, a set of shadowing rules are detected and removed from a top-bottom scope, by iteratively applying Algorithm 1 – when the decision field of the two rules is different. Let us notice that this stage of detecting and removing shadowed rules is applied before the detection and removal of proper redundant and irrelevant rules.

The resulting set of rules is then used when applying the second phase, also from a top-bottom scope. This stage is performed to detect and remove proper redundant rules, through an iterative call to Algorithm 3 (i.e., *testRedundancy*), as well as to detect and remove all the further shadowed rules remaining during the latter process. Finally, during a third phase the whole set of non-empty rules is analyzed

Algorithm 1: exclusion(B, A)

```

1  $C[condition] \leftarrow \emptyset$ ;
2  $C[shadowing] \leftarrow false$ ;
3  $C[redundancy] \leftarrow false$ ;
4  $C[irrelevance] \leftarrow false$ ;
5  $C[decision] \leftarrow B[decision]$ ;
6 forall the elements of  $A[condition]$  and  $B[condition]$  do
7   if  $((A_1 \cap B_1) \neq \emptyset$  and  $(A_2 \cap B_2) \neq \emptyset$ 
8   and ... and  $(A_p \cap B_p) \neq \emptyset)$  then
9      $C[condition] \leftarrow C[condition] \cup$ 
10     $\{(B_1 - A_1) \wedge B_2 \wedge \dots \wedge B_p,$ 
11     $(A_1 \cap B_1) \wedge (B_2 - A_2) \wedge \dots \wedge B_p,$ 
12     $(A_1 \cap B_1) \wedge (A_2 \cap B_2) \wedge (B_3 - A_3) \wedge \dots \wedge B_p,$ 
13     $\dots$ 
14     $(A_1 \cap B_1) \wedge \dots \wedge (A_{p-1} \cap B_{p-1}) \wedge (B_p - A_p)\}$ ;
15   else
16      $C[condition] \leftarrow (C[condition] \cup B[condition])$ ;
17 return  $C$ ;

```

Algorithm 2: testIrrelevance(c, r)

```

1  $z_s \leftarrow source(r)$ ;
2  $z_d \leftarrow dest(r)$ ;
3 if  $(z_s = z_d)$  and  $(\neg r[decision])$  then
4   warning ("First case of irrelevance");
5 else if  $z_s \neq z_d$  then
6    $p \leftarrow minimal\_route(z_s, z_d)$ ;
7   if  $c \notin p$  and  $(\neg r[decision])$  then
8     warning ("Second case of irrelevance");
9   else if  $(\neg empty(r[A_c]))$  and  $(\neg affects(z_d, r[A_c]))$  then
10    warning ("Third case of irrelevance");
11   else return false;
12 return true;

```

Algorithm 3: testRedundancy(R, r)

```

1  $i \leftarrow 1$ ;
2  $temp \leftarrow r$ ;
3 while  $\neg test$  and  $(i \leq count(R))$  do
4    $temp \leftarrow exclusion(temp, R_i)$ ;
5   if  $temp[condition] = \emptyset$  then
6     return true;
7    $i \leftarrow (i + 1)$ ;
8 return false;

```

Algorithm 4: intra-component-audit(c, R)

```

1 begin
2    $n \leftarrow count(R)$ ;
3   /*Phase 1*/
4   for  $i \leftarrow 1$  to  $(n - 1)$  do
5     for  $j \leftarrow (i + 1)$  to  $n$  do
6       if  $R_i[decision] \neq R_j[decision]$  then
7          $R_j \leftarrow exclusion(R_j, R_i)$ ;
8         if  $R_j[condition] = \emptyset$  then
9           warning ("Shadowing");
10         $R_j[shadowing] \leftarrow true$ ;
11   /*Phase 2*/
12   for  $i \leftarrow 1$  to  $(n - 1)$  do
13      $R_a \leftarrow \{r_k \in R \mid n \geq k > i \text{ and}$ 
14      $r_k[decision] = r_i[decision]\}$ ;
15     if testRedundancy( $R_a, R_i$ ) then
16       warning ("Redundancy");
17        $R_i[condition] \leftarrow \emptyset$ ;
18        $R_i[redundancy] \leftarrow true$ ;
19     else
20       for  $j \leftarrow (i + 1)$  to  $n$  do
21         if  $R_i[decision] = R_j[decision]$  then
22            $R_j \leftarrow exclusion(R_j, R_i)$ ;
23           if  $(\neg R_j[redundancy])$  and
24            $R_j[condition] = \emptyset$  then
25             warning ("Shadowing");
26              $R_j[shadowing] \leftarrow true$ ;
27   /*Phase 3*/
28   for  $i \leftarrow 1$  to  $n$  do
29     if  $R_i[condition] \neq \emptyset$  then
30       if testIrrelevance( $c, R_i$ ) then
31          $R_i[irrelevance] \leftarrow true$ ;
32          $r[condition] \leftarrow \emptyset$ ;
33 end

```

in order to detect and remove irrelevance, through an iterative call to Algorithm 2 (i.e., *testIrrelevance*).

Let us conclude by giving an outlook to the set of warnings send to the security officer after the execution of Algorithm 4 over the configuration of the two components shown in Figure 2.

<p>First case of irrelevance on $C_1 \{R_1\}$ Redundancy on $C_1 \{R_4\}$ Shadowing on $C_1 \{R_6\}$</p>
--

<p>Third case of irrelevance on $C_2 \{R_2\}$ Second case of irrelevance on $C_2 \{R_3\}$ Redundancy on $C_2 \{R_6\}$</p>

3.3 Correctness of the Algorithms

Lemma 1 Let $R_i : condition_i \rightarrow decision_i$ and $R_j : condition_j \rightarrow decision_j$ be two configuration rules. Then $\{R_i, R_j\}$ is equivalent to $\{R_i, R'_j\}$ where $R'_j \leftarrow exclusion(R_j, R_i)$.

Proof of Lemma 1 Let us assume that $R_i[condition] = A_1 \wedge A_2 \wedge \dots \wedge A_p$, and $R_j[condition] = B_1 \wedge B_2 \wedge \dots \wedge B_p$. If $(A_1 \cap B_1) = \emptyset$ or $(A_2 \cap B_2) = \emptyset$ or \dots or $(A_p \cap B_p) = \emptyset$ then $exclusion(R_j, R_i) \leftarrow R_j$. Hence, to prove the equivalence between $\{R_i, R_j\}$ and $\{R_i, R'_j\}$ is trivial in this case.

Let us now assume that $(A_1 \cap B_1) \neq \emptyset$ and $(A_2 \cap B_2) \neq \emptyset$ and \dots and $(A_p \cap B_p) \neq \emptyset$. If we apply rules $\{R_i, R_j\}$ where R_i comes before R_j , then rule R_j applies to a given packet if this packet satisfies $R_j[condition]$ but not $R_i[condition]$ (since R_i applies first). Therefore, notice that $R_j[condition] - R_i[condition]$ is equivalent to $(B_1 - A_1) \wedge B_2 \wedge \dots \wedge B_p$ or $(A_1 \cap B_1) \wedge (B_2 - A_2) \wedge \dots \wedge B_p$ or $(A_1 \cap B_1) \wedge (A_2 \cap B_2) \wedge (B_3 - A_3) \wedge \dots \wedge B_p$ or \dots $(A_1 \cap B_1) \wedge \dots \wedge (A_{p-1} \cap B_{p-1}) \wedge (B_p - A_p)$, which corresponds to $R'_j = exclusion(R_j, R_i)$. This way, if R_j applies to a given packet in $\{R_i, R_j\}$, then rule R'_j also applies to this packet in $\{R_i, R'_j\}$. Conversely, if R'_j applies to a given packet in $\{R_i, R'_j\}$, then this means this packet satisfies $R_j[condition]$ but not $R_i[condition]$. So, it is clear that rule R_j also applies to this packet in $\{R_i, R_j\}$. Since in Algorithm 1 $R'_j[decision]$ becomes $R_j[decision]$, this enables to conclude that $\{R_i, R_j\}$ is equivalent to $\{R_i, R'_j\}$. \square

Theorem 2 Let R be a set of configuration rules and let $Tr(R)$ be the resulting rules obtained by applying Algorithm 4 to R . Then R and $Tr(R)$ are equivalent.

Proof of Theorem 2 Let $Tr'_1(R)$ be the set of rules obtained after applying the first phase of Algorithm 4. Since $Tr'_1(R)$ is derived from rule R by applying $exclusion(R_j, R_i)$ to some rules R_j in R , it is straightforward, from Lemma 1, to conclude that $Tr'_1(R)$ is equivalent to R .

Let us now move to the second phase, and let us consider a rule R_i such that $testRedundancy(R_i)$ (cf. Algorithm 3) is *true*. This means that $R_i[condition]$ can be derived by conditions of a set of rules S with the same decision and that come after in order than rule R_i . Since every rule R_j with a decision different from the one of rules in S has already been excluded from rules of S in the first phase of the Algorithm, we can conclude that rule R_i is definitely redundant and can be removed without changing the component configuration. This way, we conclude that Algorithm 4 preserves equivalence in this case. On the other hand, if $testRedundancy(R_i)$ is *false*, then transformation consists in applying function $exclusion(R_j, R_i)$ to some rules R_j which also preserves equivalence.

Similarly, and once in the third phase, let us consider a rule R_i such that $testIrrelevance(c, R_i)$ is *true*. This means that this rule matches traffic that will never cross component c , or that is irrelevant for the component's configuration. So, we can remove R_i from R without changing such a configuration. Thus, in this third case, as in the other two cases, $Tr'(R)$ is equivalent to $Tr'_1(R)$ which, in turn, is equivalent to R . \square

Lemma 3 *Let $R_i : condition_i \rightarrow decision_i$ and $R_j : condition_j \rightarrow decision_j$ be two configuration rules. Then rules R_i and R'_j , where $R'_j \leftarrow exclusion(R_j, R_i)$ will never simultaneously apply to any given packet.*

Proof of Lemma 3 Notice that rule R'_j only applies when rule R_i does not apply. Thus, if rule R'_j comes before rule R_i , this will not change the final decision since rule R'_j only applies to packets that do not match rule R_i . \square

Theorem 4 *Let R be a set of configuration rules and let $Tr(R)$ be the resulting rules obtained by applying Algorithm 4 to R . Then the following statements hold: (1) Ordering the rules in $Tr(R)$ is no longer relevant; (2) $Tr(R)$ is completely free of anomalies.*

Proof of Theorem 4 For any pair of rules R_i and R_j such that R_i comes before R_j , R_j is replaced by a rule R'_j obtained by recursively replacing R_j by $exclusion(R_j, R_k)$ for any $k < j$.

Then, by recursively applying Lemma 3, it is possible to commute rules R'_i and R'_j in $Tr(R)$ without changing the policy.

Regarding the second statement – $Tr(R)$ is completely free of anomalies – notice that, in $Tr(R)$, each rule is independent of all other rules. Thus, if we consider a rule R_i in $Tr(R)$ such that $R_i[condition] \neq \emptyset$, then this rule will apply to any packet that satisfies $R_i[condition]$, i.e., it is not shadowed.

On the other hand, rule R_i is not redundant because if we remove this rule, since this rule is the only one that applies to packets that satisfy $R_i[condition]$, then configuration of the component will change if we remove rule R_i from $Tr(R)$.

Finally, and after the execution of Algorithm 4 over the initial set of configuration rules, one may verify that for each rule R_i in $Tr(R)$ the following conditions hold: (1) $s = z_1 \cap source(r) \neq \emptyset$ and $d = z_2 \cap dest(r) \neq \emptyset$ such that $z_1 \neq z_2$ and component c is in $minimal_route(z_1, z_2)$; (2) if $A_c = attack_category(R_i) \neq \emptyset$, the predicate $affects(A_c, z_2)$ becomes *true*. Thus, each rule R_i in $Tr(R)$ is not irrelevant. \square

3.4 Default policies

Each component implements a positive (i.e., close) or negative (i.e., open) default policy. In the positive policy, the default policy is to *alert* or to *deny* a packet when any configuration rule applies. By contrast, the negative policy will *accepts* or *pass* a packet when no rule applies.

After rewriting the rules with our algorithms, we can actually remove every rule whose decision is *pass* or *accept* if the default policy of this component is negative (else this rule is redundant with the default policy) and similarly we can remove every rule whose decision is *deny* or *alert* if the default policy is positive. Thus, we can consider that our proposed algorithms generate a configuration that only contains positive rules if the component default policy is negative, and negative rules if the default policy is positive.

4 Performance Evaluation

In this section, we present an evaluation of the performance of MIRAGE (which stands for MISconfigURAtion manaGER), a software prototype that implements the algorithms presented in sections 3. MIRAGE has been developed using PHP, a scripting language that is especially suited for web services development and can be embedded into HTML for the construction of client-side GUI based applications [3]. MIRAGE can be locally or remotely executed by using a HTTP server (e.g., Apache server over UNIX or Windows setups) and a web browser.

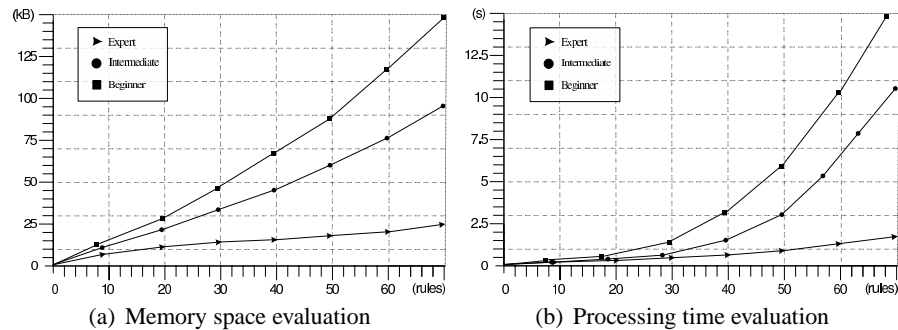


Fig. 3. Memory and processing time evaluation.

We evaluated our algorithms through a set of experiments over an IPv4 network. The topology for this network consisted of a single firewall based on Netfilter [13], and a single NIDS based on Snort [12] – both of them connected to three different zones with more than 50 hosts. The whole of these experiments were carried out

on an Intel-Pentium M 1.4 GHz processor with 512 MB RAM, running Debian GNU/Linux 2.6.8, and using Apache/1.3 with PHP/4.3 configured.

During our experiments, we measured the memory space and the processing time needed to perform Algorithm 4 over several sets of IPv4 filtering and alerting policies for the two IPv4 networks, according to the three following security officer profiles: beginner, intermediate, and expert – where the probability to have overlaps between rules increases from 5% to 90%. The results of these measurements are plotted in Figure 3(a) and Figure 3(b). Although the plots reflect strong memory and process time requirements, we consider they are reasonable for off-line analysis, since it is not part of the critical performance of an alerting or filtering component.

5 Related Work

A first approach to get a configuration free of errors is by applying a formal model to express the network policy. In [4], for example, we presented a model with this purpose. This way, a set of configuration rules, whose syntax is specific to a given component, may be generated using a transformation language.

The proposals in [1, 6, 7, 2], provide means to directly manage the discovery of anomalies from the components' configuration. For instance, the authors in [1] consider that, in a configuration set, two rules are in conflict when the first rule in order matches some packets that match the second rule, and the second rule also matches some of the packets that match the first rule. This approach is very limited since it just detects a particular case of wrongly defined rules in a single configuration, i.e., just ambiguity within the set of rules is detected.

In [6], two new cases of anomalies are considered. First, a rule R_j is defined as backward redundant iff there exists another rule R_i with higher priority in order such that all the packets that match rule R_j also match rule R_i . Second, a rule R_i is defined as forward redundant iff there exists another rule R_j with the same decision and less priority in order such that the following conditions hold: (1) all the packets that match R_i also match R_j ; (2) for each rule R_k between R_i and R_j , and that matches all the packets that also match rule R_i , R_k has the same decision as R_i . We consider this approach as incomplete, since it does not detect all the possible cases of anomalies defined in this paper. For instance, given the set of rules shown in Figure 4(a), since R_2 comes after R_1 , rule R_2 only applies over the interval [51, 70] – i.e., R_2 is redundant. Their approach, however, cannot detect the redundancy of rule R_2 within this setup.

Another similar approach is presented in [2]. Again, and even though the efficiency of their proposed discovering algorithms and techniques is very promising, we con-

$R_1 : s \in [10, 50] \rightarrow deny$	$R_1 : s \in [10, 50] \rightarrow accept$
$R_2 : s \in [40, 70] \rightarrow accept$	$R_2 : s \in [40, 90] \rightarrow accept$
$R_3 : s \in [50, 80] \rightarrow accept$	$R_3 : s \in [30, 80] \rightarrow deny$
(a) Set of rules A	(b) Set of rules B

Fig. 4. Example of some firewall configurations.

sider this approach not complete since, given a misconfigured component, their detection algorithms could not detect all the possible errors. For example, given the set of rules shown in Figure 4(b) their approach cannot detect that rule R_3 will be never applied due to the union of rules R_1 and R_2 .

6 Conclusions

In this paper we presented an audit process to set the configuration of both *firewalls* and *network intrusion detection systems* (NIDSs) free of anomalies. Our audit process is based on the existence of relationships between the condition attributes of the configuration rules of those network security components, such as coincidence, disjunction, and inclusion. Then, our proposal uses a transformation process which derives from an initial set of rules – potentially misconfigured – to an equivalent one which is completely free of anomalies.

We also presented in this paper a network model to determine whether the network traffic that matches a given configuration rule, may or may not cross the component configured by such a rule, as well as other network properties. Thanks to this model, our approach best defines all the set of anomalies studied in the related work, and it reports, moreover, a new anomaly case not reported, as defined in this paper, in none of the other approaches.

Some advantages of our approach are the following. First, our transformation process verifies that the resulting rules are completely independent between them. Otherwise, each rule considered as useless during the process is reported to the security officer, in order to verify the correctness of the whole process. Second, we can perform a second rewriting of rules, generating a configuration that only contains positive rules if the component default policy is negative, and negative rules if the default policy is positive. Third, the elimination of alerting rules during the audit process helps to reduce future false positive events alerted by a NIDS.

Regarding a possible increase of the initial number of rules, due to the applying of our algorithms, it is only significant whether the associated parsing algorithm of the component depends on the number of rules. In this case, an increase in such a parameter may degrade the performance of the component. Nonetheless, this is not a disadvantage since the use of a parsing algorithm independent of the number of

rules becomes the best solution as much for our proposal as for the current deployment of network technologies. The set pruning tree algorithm is a proper example, because it only depends on the number and size of attributes to be parsed, not the number of rules [11].

The implementation of our approach in a software prototype demonstrate the practicability of our work. We shortly discussed this implementation, based on a scripting language [3], and presented an evaluation of its performance. Although these experimental results show that our algorithms have strong requirements, we believe that they are reasonable for off-line analysis, since it is not part of the critical performance of the audited component.

Acknowledgments

This work was supported by funding from the French ministry of research, under the *ACI DESIRS* project, the Spanish Government project *TIC2003-02041*, and the Catalan Government grants *2003FI126* and *2005BE77*.

References

1. Adishesu, H., Suri, S., and Parulkar, G. (2000). Detecting and Resolving Packet Filter Conflicts. *19th Annual Joint Conference of the IEEE Computer and Communications Societies*.
2. Al-Shaer, E. S., Hamed, H. H., and Masum, H. (2005). Conflict Classification and Analysis of Distributed Firewall Policies In *IEEE Journal on Selected Areas in Communications*, 1(1).
3. Castagnetto, J. et al. (1999). *Professional PHP Programming*.
4. Cuppens, F., Cuppens-Bouahia, N., Sans, T. and Mieke, A. (2004). In *Second Workshop on Formal Aspects in Security and Trust*. A formal approach to specify and deploy a network security policy. In *Second Workshop on Formal Aspects in Security and Trust*, 203–218.
5. Cuppens, F., Cuppens-Bouahia, N., and García-Alfaro, J. (2005). Detection and Removal of Firewall Misconfiguration. In *2005 International Conference on Communication, Network and Information Security*. 154–162.
6. Gupta, P. (2000). *Algorithms for Routing Lookups and Packet Classification*. PhD Thesis, Department of Computer Science, Stanford University.
7. Liu, A. X. and Gouda, M. G. (2005). Complete Redundancy Detection in Firewalls. In *Proceedings of 19th Annual IFIP Conference on Data and Applications Security*, 196–209.
8. MITRE Corp. Common Vulnerabilities and Exposures. [Online]. Available from: <http://cve.mitre.org/>
9. Northcutt, S. (2002). *Network Intrusion Detection: An analyst's Hand Book*. New Riders Publishing, third edition edition.
10. Open Security Foundation. Open Source Vulnerability Database. [Online]. Available from: <http://osvdb.org/>
11. Paul, O., Laurent, M., and Gombault, S. (2000). A full bandwidth ATM Firewall. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*.
12. Roesch, M. (1999), Snort: lightweight intrusion detection for networks. In *13th USENIX Systems Administration Conference*, Seattle, WA.
13. Welte, H., Kadlecik, J., Josefsson, M., McHardy, P., and et al. The netfilter project: firewalling, nat and packet mangling for linux 2.4x and 2.6.x. [Online]. Available from: <http://www.netfilter.org/>