Detection and Removal of Firewall Misconfiguration

Frederic Cuppens, Nora Boulahia Cuppens, Joaquin Garcia-alfaro

DETECTION AND REMOVAL OF FIREWALL MISCONFIGURATION

Frédéric Cuppens Nora Cuppens-Boulahia Joaquín García-Alfaro

GET/ENST-Bretagne,
2, rue de la Châtaigneraie,
35576 Cesson Sévigné - France
{frederic.cuppens,nora.cuppens}@enst-bretagne.fr

dEIC/UAB,
Edifici Q, Campus de Bellaterra,
08193, Bellaterra, Barcelona - Spain
joaquin.garcia@uab.es

ABSTRACT

To police network traffic, firewalls must be configured with a set of filtering rules. The existence of errors in this set is very likely to degrade the network security policy. The management of these configuration errors is a serious and complex problem to solve. In this paper, we present a set of algorithms to manage rules that never apply or are redundant in a firewall configuration. Our approach is based on the analysis of relationships between the set of filtering rules. Then, a subsequent rewriting of rules will derive from an initial firewall setup to an equivalent one completely free of errors. At the same time, the algorithms will detect both shadowed and redundant rules in the initial firewall configuration.

KEY WORDS

Network Security, Firewalls, Filtering Rules, Redundancy, Shadowing of Rules

1 Introduction

Many companies and organizations use firewalls to segment access control within their own networks. Firewalls are typically deployed to filter traffic between *trusted* and *untrusted* zones of corporate networks, as well as to police their incoming and outcoming interaction with other networks – e.g., the Internet[1].

Firewalls, as many other network security components, must to be configured with a list of rules (e.g., the set of filtering rules shown in Table 1). Each rule typically specifies a $decision$ (e.g., $accept$ or $deny$) that applies to a set of $condition$ attributes, such as protocol, source, destination, service ports, and so on. For our work, we define a filtering 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, $decision_i$ is a boolean expression in $\{accept, deny\}$[2], and $\{condition_i\}$ is a conjunctive set of condition attributes such that $\{condition_i\}$ equals $A_1 \wedge A_2 \wedge ... \wedge A_p$ – p being the number of condition attributes of the given set of filtering rules.

To solve conflicts when processing packages, most firewall implementations use a *first matching* strategy through the ordering of rules (e.g. the *order* field shown in Table 1). Hence, each packet processed by the firewall is mapped to the decision of the highest priority rule.

This strategy introduces, however, firewall rule conflicts that result in non-firing rules (*shadowing*) or rules that have no effect (*redundancy*). We define these two general cases of firewall misconfiguration as follows.

Definition 1.1 *Let R be a set of filtering rules. Then R has **shadowing** iff there exists at least one filtering rule, R_i in R, which 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.*

Definition 1.2 *Let R be a set of filtering rules. Then R has **redundancy** iff there exists at least one filtering rule, R_i in R, such that the following conditions hold: (1) R_i is not shadowed by any other rule; (2) when removing R_i from R, the filtering result does not change.*

The detection and removal of both redundancy and shadowing is a serious problem which must to be solved, since a misconfigured set of filtering rules, if not handled correctly, is very likely to lead to a inefficient and weak security policy – since packets could be subject to the wrong actions. Although the proper managing of this configuration errors is starting to get currently a significant amount of work, such as [1, 7, 10, 2, 4], just few approaches seem to manage acceptable results.

[1] 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.

[2] The $decision$ field may also be a combination of both $accept$ and $deny$ together with some other options such as a logging or jump options. For reasons of clarity we assume that just accept and deny are proper values.

order	condition					decision
	(p)rotocol	(s)ource	(sP)ort	(d)estination	(dP)ort	
1	any	xxx.xxx.xxx.[010,050]	any	xxx.xxx.xxx.xxx	any	deny
2	any	xxx.xxx.xxx.[040,090]	any	xxx.xxx.xxx.xxx	any	accept
3	any	xxx.xxx.xxx.[060,100]	any	xxx.xxx.xxx.xxx	any	accept
4	any	xxx.xxx.xxx.[030,080]	any	xxx.xxx.xxx.xxx	any	deny
5	any	xxx.xxx.xxx.[001,070]	any	xxx.xxx.xxx.xxx	any	accept

Table 1. Example of a set of filtering rules with five condition attributes.

In this paper, we present a set of algorithms for the managing of both redundancy and shadowing of rules. Our main objective is the following. Given a specific firewall setup, we want to analyze the existing firewall configuration to check whether there is errors in such a configuration, i.e., the set of filtering rules presents shadowing or redundancy as defined above.

Our approach is based on the relationships between the filtering rules' parameters: coincidence, disjunction and inclusion. We use a rule transformation process that derive from a set of filtering rules to an equivalent and valid one that is completely free of both shadowing and redundancy. The advantages of our proposal are the following.

After rewriting the rules, one can verify that each redundant or shadowed rule – considered as useless during the audit process – will be removed from the initial set of filtering rules. When such a detection occurs, the discovering process will provide an evidence of error to the administration console. This way, the security officer in charge of the network can check from the initial specification, in order to verify the correctness of the whole process.

Furthermore, since the resulting rules are disjoint, the ordering of rules is no longer relevant. Hence, one can perform a second transformation in a positive or negative manner. The first – when generating only permissions – can be used in a closed policy whereas the latter – when generating only prohibitions – can be used in case of an open policy. After performing this second rewriting, the security officer will have a clear view of the accepted traffic (in the case of positive rewriting) or the rejected traffic (in the case of negative rewriting).

The rest of this paper is organized as follows: We start in Section 2 with an analysis of some related work. In Section 3 and Section 4, we formally present our algorithms, and introduce some examples and demonstrations to validate the correctness of our approach. We close with conclusions and future work in Section 5.

2 Related Work

A first approach to get a firewall configuration free of errors is by applying a formal security model to express the network security policy. In [4], for example, a formal model is presented with this purpose. This way, a set of filtering rules, whose syntax is specific to a given firewall, may be generated using a transformation language. Nonetheless, this approach is not enough to ensure that the firewall configuration is completely free of errors.

Some other proposals, such as [1, 7, 2, 10], provide means to directly manage misconfiguration. The authors in [1], for instance, 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 packets that match the first rule. This approach is very limited since it does not detect what we consider *serious misconfiguration errors* – the two general cases defined in Section1. What they detect is just a particular case of wrongly defined rules which cause ambiguity in the firewall configuration, and that is more efficiently defined as a combination of both redundancy and shadowing.

In [7], two new cases of misconfiguration are considered. First, a rule R_j is defined as backward redundant if and only if 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. On the other hand, a rule R_i is defined as forward redundant if and only if 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 than R_i.

Although this approach seems to head in the right direction, we consider our definitions (cf. Section1, Def. 1.1 and Def. 1.2) simpler and more general, because all possible backward and forward redundant rules are specific cases of both redundancy and shadowing, but not vice versa. For instance, given the following set of rules:

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : s \in [40, 70] \rightarrow accept$$
$$R_2 : s \in [50, 80] \rightarrow accept$$

their detection proposal, as defined above, cannot detect the redundancy of rule R_2. Thus, we point out this work as incomplete.

Firewall Builder [10], a well known support tool developed to assist administrators in their task of configuring firewalls, also provides detection of misconfiguration in the set of filtering rules. Nevertheless, this discovering mechanism only detects trivial equality or inclusion between the

filtering rules' parameters. We checked out that more complex configuration errors are unfortunately not detected by this tool.

To our best knowledge, the authors in [2] propose the most efficient set of techniques and algorithms to detect redundancy and shadowing in different firewall configuration setups. In addition to the discovery process, their approach also attempts an optimal insertion of arbitrary rules into an existing configuration, through a tree based representation of the filtering criteria.

Nonetheless, and even though the efficiency of their proposed discovering algorithms and techniques is very promising, we also consider this approach as incomplete. On the one hand, their approach is too weak since, given a misconfigured firewall, their discovering algorithms could not detect all the possible errors. For example, given the following set of rules:

$$R_1 : s \in [10, 50] \rightarrow accept$$
$$R_2 : s \in [40, 90] \rightarrow accept$$
$$R_3 : s \in [30, 80] \rightarrow deny$$

their approach cannot detect the shadowing over rule R_3 due to the union of rules R_1 and R_2. Furthermore, they do not cover, intentionally, an automatic rewriting of rules to correct the discovered errors. This way, it is the security officer who should perform the final changes.

Summing up, we believe that none of the identified related work provides a complete discovering of both redundancy and shadowing of rules – which are the cases we consider *serious errors* within firewalls configurations – as well as a proper handling of such a misconfiguration.

3 Detection Process

As pointed out in Section 1, our main goal is the discovering of both shadowing and redundancy inside an initial set of filtering rules R. Such a detection 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 firewall configuration.

The data for the process is the following. A set of rules R as a linked-list[3]of initial size n, where n equals $count(R)$, and where each element is an associative arraywith the strings $condition$, $decision$, $shadowing$, and $redundancy$ as keys to access each necessary value. In turn, each element $R_i[condition]$ is an indexed arrayof size p containing the set of conditions of each rule; each element $R_i[decision]$ is a boolean variable whose values are in $\{accept, deny\}$; each element $R_i[shadowing]$ is a boolean variable in $\{true, false\}$; each element $R_i[redundancy]$ is another boolean variable in $\{true, false\}$. Both shadowing and redundancy variables of each rule are initialized to $false$.

To simplify, we split the whole detection process and the removal of misconfiguration in two different processes. Thus, we define a main detection function (Algorithm 1), whose input is the initial set of filtering rules, R, and an auxiliary function (Algorithm 2) 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 second algorithm (cf. Algorithm 2), we use the notation A_i as an abbreviation of the variable $A[condition][i]$, an the notation B_i as an abbreviation of the variable $B[condition][i]$ – where i in $[1, p]$.

Algorithm 1: `detection`(R)

```
for i ← 1 to (count(R) − 1) do
    for j ← (i + 1) to count(R) do
        R_j ← exclusion (R_j,R_i);
        if R_j[condition] = ∅
        then R_j[shadowing] ← true;
    end
end
```

Algorithm 2: `exclusion`(B,A)

```
C[condition] ← ∅;
C[decision] ← B[decision];
C[shadowing] ← false;
C[redundancy] ← false;
forall the elements of A[condition] and B[condition] do
    if ((A_1 ∩ B_1) ≠ ∅ and (A_2 ∩ B_2) ≠ ∅ and ...
    ... and (A_p ∩ B_p) ≠ ∅)
    then
        C[condition] ← C[condition] ∪
        {(B_1 − A_1) ∧ B_2 ∧ ... ∧ B_p,
        (A_1 ∩ B_1) ∧ (B_2 − A_2) ∧ ... ∧ B_p,
        (A_1 ∩ B_1) ∧ (A_2 ∩ B_2) ∧ (B_3 − A_3) ∧ ... ∧ B_p,
        ...
        (A_1 ∩ B_1) ∧ ... ∧ (A_{p−1} ∩ B_{p−1}) ∧ (B_p − A_p)};
    else
        C[condition] ←
        (C[condition] ∪ B[condition]);
    end
end
return C;
```

We recall that the output of the main detection function (cf. Algorithm 1) is the set which results as a transformation of the initial set of filtering rules R. This new set is equivalent to the initial one, R, and all its rules are completely disjoint. Therefore, the resulting set is free of both redundancy and shadowing of rules, as well as any other possible configuration error – such as correlation, generalization, and irrelevance [2].

[3]We assume one can access a linked-list through the operator R_i, where i is the relative position regarding the initial list size – $count(R)$. We also assume one can 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.

3.1 Applying the Algorithms

This section gives a short outlook on applying our algorithms over some representative examples. Let us start by applying function *exclusion* over a set of two rules R_i and R_j, each one of them with two condition attributes – (s)ource and (d)estination – and where rule R_j has less priority in order than rule R_i. In this first example,

$$R_i[condition] = (s \in [80, 100]) \wedge (d \in [1, 50])$$
$$R_j[condition] = (s \in [1, 50]) \wedge (d \in [1, 50])$$

since $(s \in [1, 50]) \cap (s \in [80, 100]) = \emptyset$, the condition attributes of rules R_i and R_j are completely independent. Thus, the applying of $exclusion(R_j, R_i)$ is equal to $R_j[condition]$.

The following three examples show the same execution over a set of condition with different cases of conflict. A first case is the following,

$$R_i[condition] = (s \in [1, 60]) \wedge (d \in [1, 30])$$
$$R_j[condition] = (s \in [1, 50]) \wedge (d \in [1, 50])$$

where there is a main overlap of attribute s from $R_i[condition]$ which completely excludes the same attribute on $R_j[condition]$. Then, there is a second overlap of attribute d from $R_i[condition]$ which partially excludes the range $[1, 30]$ into attribute d of $R_j[condition]$, which becomes d in $[31, 50]$. This way, $exclusion(R_j, R_i) \leftarrow \{(s \in [1, 50]) \wedge (d \in [31, 50])\}$[4]. In this second example,

$$R_i[condition] = (s \in [1, 60]) \wedge (d \in [20, 30])$$
$$R_j[condition] = (s \in [1, 50]) \wedge (d \in [1, 50])$$

there is two simple overlaps of both attributes s and d from $R_i[condition]$ to $R_j[condition]$, such that $exclusion(R_j, R_i)$ becomes $\{(s \in [1, 50]) \wedge (d \in [1, 19]), (s \in [1, 50]) \wedge (d \in [31, 50])\}$. A more complete example is the following,

$$R_i[condition] = (s \in [10, 40]) \wedge (d \in [20, 30])$$
$$R_j[condition] = (s \in [1, 50]) \wedge (d \in [1, 50])$$

where $exclusion(R_j, R_i)$ becomes $\{(s \in [1, 9]) \wedge (d \in [1, 50]), (s \in [41, 50]) \wedge (d \in [1, 50]), (s \in [10, 40]) \wedge (d \in [1, 19]), (s \in [10, 40]) \wedge (d \in [31, 50])\}$.

Regarding a full exclusion, let us now show the following example,

$$R_i[condition] = (s \in [1, 60]) \wedge (d \in [1, 60])$$
$$R_j[condition] = (s \in [1, 50]) \wedge (d \in [1, 50])$$

where the set of condition attributes of rule R_i completely excludes the ones of rule R_j. Then, the applying of $exclusion(R_j, R_i)$ becomes an empty set (i.e., $\{\emptyset, \emptyset\} = \emptyset$). Hence, on a further execution of Algorithm 1 the shadowing field of rule R_j (initialized as $false$ by default) would become $true$ (i.e., $R_j[shadowing] \leftarrow true$).

To conclude, let us show a complete applying over the set of filtering rules based on Table 1, where the number of condition attributes, p, is just one.

$/ * motivation\ example * /$
$R_1 : s \in [10, 50] \rightarrow deny$
$R_2 : s \in [40, 90] \rightarrow accept$
$R_3 : s \in [60, 100] \rightarrow accept$
$R_4 : s \in [30, 80] \rightarrow deny$
$R_5 : s \in [1, 70] \rightarrow accept$

$/ * step\ 1 * /$
$R_1 : s \in [10, 50] \rightarrow deny$
$R_2 : s \in [51, 90] \rightarrow accept$
$R_3 : s \in [60, 100] \rightarrow accept$
$R_4 : s \in [51, 80] \rightarrow deny$
$R_5 : \{s \in [1, 9], s \in [51, 70]\} \rightarrow accept$

$/ * step\ 2 = step\ 3 = step\ 4 * /$
$R_1 : s \in [10, 50] \rightarrow deny$
$R_2 : s \in [51, 90] \rightarrow accept$
$R_3 : s \in [91, 100] \rightarrow accept$
$R_4 : \emptyset \rightarrow deny$
$R_5 : s \in [1, 9] \rightarrow accept$

$/ * resulting\ rules * /$
$R_1 : s \in [10, 50] \rightarrow deny$
$R_2 : s \in [51, 90] \rightarrow accept$
$R_3 : s \in [91, 100] \rightarrow accept$
$R_5 : s \in [1, 9] \rightarrow accept$

$/ * warnings * /$
$R_4[shadowing] = true$

3.2 Correctness of the Algorithms

Definition 3.1 *Let R be a set of filtering rules and let $Tr(R)$ be the resulting filtering rules obtained by applying Algorithm 1 to R.*

Lemma 3.2 *Let $R_i : condition_i \rightarrow decision_i$ and $R_j : condition_j \rightarrow decision_j$ be two filtering rules. Then $\{R_i, R_j\}$ is equivalent to $\{R_i, R'_j\}$ where $R'_j \leftarrow exclusion(R_j, R_i)$.*

Theorem 3.3 *Let R be a set of filtering rules and let $Tr(R)$ be the resulting filtering rules obtained by applying Algorithm 1 to R. Then R and $Tr(R)$ are equivalent.*

Lemma 3.4 *Let $R_i : condition_i \rightarrow decision_i$ and $R_j : condition_j \rightarrow decision_j$ be two filtering 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.*

[4] We do not show the first empty set corresponding to the first overlap. If shown, the result should become as follows: $exclusion(R_j, R_i) \leftarrow \{\emptyset, (s \in [1, 50]) \wedge (d \in [31, 50])\}$.

Theorem 3.5 *Let R be a set of filtering rules and let $Tr(R)$ be the resulting filtering rules obtained by applying Algorithm 1 to R. Then ordering the rules in $Tr(R)$ is no longer relevant.*

Theorem 3.6 *Let R be a set of filtering rules and let $Tr(R)$ be the resulting filtering rules obtained by applying Algorithm 1 to R. Then $Tr(R)$ is free from both shadowing and redundancy.*

For space limitation reasons, we move the correctness proofs of this section to Appendix A.

4 Complete Detection

Up to now, the result of Algorithm 1 offers a set of filtering rules, $Tr(R)$, equivalent to an initial set of rules R, and completely free of any possible relation between its rules. Nevertheless, there is a limitation on such an algorithm regarding the reporting of redundancy – just the existence of shadowing is reported to the security officer. Therefore, we need to modify this algorithm in order to also detect redundancy in R.

The purpose of this section is to solve this limitation, by presenting a second manner to completely discover both shadowing and redundancy errors into the initial set of filtering rules, R, based on the techniques and results previously shown in Section 3.

The reporting of redundancy is much more complex than the task of reporting shadowing. To properly overcome this complexity, we first divide the whole process in two different algorithms (Algorithm 3 and Algorithm 4).

The first algorithm (cf. Algorithm 3) is a boolean function in $\{true, false\}$, which, in turn, apply the transformation *exclusion* (cf. Section 3, Algorithm 2) over a set of filtering rules to check whether the rule obtained as a parameter is potentially redundant.

The second algorithm (cf. Algorithm 4) performs the whole process of detecting and removing both redundancy and shadowing, and is also split in two different phases. During the first phase, a potential set of redundant rules is calculated from a top-bottom scope, by iteratively applying Algorithm 3. Such a process is applied during the stage of detecting and removing shadowed rules, i.e., before the detection and removal of proper redundant rules. This information is then used when applying the second phase, from a bottom-top scope. This stage is performed to detect and remove proper redundant rules, as well as to detect and remove all the further shadowed rules resulting during the process. As a result of the whole execution, the initial set of rules, R, is transformed into an equivalent set, $Tr(R)$, whose rules are completely disjoint. Furthermore, all the discovery of both shadowing and redundancy is reported to the security officer, who may verify the correctness of the whole process.

Algorithm 3: `testRedundancy`(R,i)

```
test ← false;
j ← (i + 1);
temp ← R_i;
while ¬test and (j ≤ count(R)) do
    if temp[decision] = R_j[decision] then
        temp ← exclusion(temp, R_j);
        if temp[condition] = ∅ then
            test ← true;
        end
    end
    j ← (j + 1);
end
return test;
```

Algorithm 4: `completeDetection`(R)

```
/* Phase 1 */
for i ← 1 to (count(R) − 1) do
    if testRedundancy (R, i) then
        R_i[redundancy] ← true;
    end
    if R_i[redundancy] then
        for j ← (i + 1) to count(R) do
            if R_i[decision] ≠ R_j[decision] then
                R_j ← exclusion (R_j,R_i);
            if R_j[condition] = ∅ then
                R_j[shadowing] ← true;
        end
    else
        for j ← (i + 1) to count(R) do
            R_j ← exclusion (R_j,R_i);
            if R_j[condition] = ∅ then
                R_j[shadowing] ← true;
        end
    end
end
/* Phase 2 */
for i ← (count(R) − 1) to 1 do
    if R_i[redundancy] then
        if testRedundancy (R, i) then
            R_i[condition] ← ∅;
        else
            R_i[redundancy] ← false;
            for j ← (i + 1) to count(R) do
                if R_i[decision]=R_j[decision] then
                    R_j ←exclusion (R_j,R_i);
                if R_j[condition] = ∅ then
                    R_j[shadowing] ← true;
            end
        end
    end
end
```

4.1 Applying the Algorithms

In this section we give an outlook on the full execution of the extended algorithms (Algorithm 3 and Algorithm 4) over a set of filtering rules based on Table 1, where the number of condition attributes, p, is just one.

$$/ * motivation\ example * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : s \in [40, 90] \rightarrow accept$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_4 : s \in [30, 80] \rightarrow deny$$
$$R_5 : s \in [1, 70] \rightarrow accept$$

Then, we begin by showing the initial step within the first phase of Algorithm 4, where i equals 1, and applied over the previous set of filtering rules. Let us notice that on this first step, the execution of function *testRedundancy* (cf. Algorithm 3), with rule R_1 as parameter, becomes $false$. Thus, the result of this first step is the following:

$$/ * phase\ 1, step\ 1, i = 1 * /$$
$$/ * testRedundancy(R_1) = false * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : s \in [51, 90] \rightarrow accept$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_4 : s \in [51, 80] \rightarrow deny$$
$$R_5 : s \in \{[1, 9], [51, 70]\} \rightarrow accept$$

Let us now move to the second step, with i equals 2. In this step, rule R_4 disappears since: (1) the result of applying function *testRedundancy* with rule R_2 as parameter becomes $true$; (2) the union of condition attributes from rules R_1 and R_2 completely excludes the condition attribute of rule R_4. Hence, rule R_4, which is shadowed by the combination of rules R_1 and R_2, becomes and empty set. Therefore, the status field $shadowing$ of rule R_4, $R_4[shadowing]$, switches its value to $true$:

$$/ * phase\ 1, step\ 2, i = 2 * /$$
$$/ * testRedundancy(R_2) = true * /$$
$$/ * R_4[shadowing] = true * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : s \in [51, 90] \rightarrow accept$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_4 : \emptyset \rightarrow deny$$
$$R_5 : s \in \{[1, 9], [51, 70]\} \rightarrow accept$$

At the end of the third step of this first phase, the applying of function *testRedundancy*, with rule R_3 as parameter, becomes $false$:

$$/ * phase\ 1, step\ 3, i = 3 * /$$
$$/ * testRedundancy(R_3) = false * /$$
$$/ * R_4[shadowing] = true * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : s \in [51, 90] \rightarrow accept$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_4 : \emptyset \rightarrow deny$$
$$R_5 : s \in \{[1, 9], [51, 59]\} \rightarrow accept$$

Once finished the first phase and running over the step where i equals 2, we notice that: (1) the applying of *testRedundancy* with R_2 as parameter becomes $true$; (2) the union of condition attributes of rules R_3 and R_5 enables the redundancy of rule R_2. Hence, the status field $redundancy$ of rule R_2, $R_2[redundancy]$, switches its value to $true$:

$$/ * phase\ 2, step\ 4, i = 2 * /$$
$$/ * testRedundancy(R_2) = true * /$$
$$/ * R_2[redundancy] = true * /$$
$$/ * R_4[shadowing] = true * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_2 : \emptyset \rightarrow accept$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_4 : \emptyset \rightarrow deny$$
$$R_5 : s \in \{[1, 9], [51, 59]\} \rightarrow accept$$

For reasons of clarity, we do not show the rest of the execution, since the resulting set of filtering rules does not modify from the previous one, which is the following:

$$/ * resulting\ rules * /$$

$$R_1 : s \in [10, 50] \rightarrow deny$$
$$R_3 : s \in [60, 100] \rightarrow accept$$
$$R_5 : s \in \{[1, 9], [51, 59]\} \rightarrow accept$$

To conclude, let us recall that the following two warnings will notice the security officer to the discovering of both shadowing and redundancy errors, in order to verify the correctness of the whole detection and transformation process:

$$/ * warnings * /$$

$$R_2[redundancy] = true$$
$$R_4[shadowing] = true$$

4.2 Correctness of the Algorithms

Theorem 4.1 *Let R be a set of filtering rules and let $Tr'(R)$ be the resulting filtering rules obtained by applying Algorithm 4 to R. Then R and $Tr'(R)$ are equivalent.*

Theorem 4.2 *Let R be a set of filtering rules and let $Tr'(R)$ be the resulting filtering rules obtained by applying Algorithm 4 to R. Then ordering the rules in $Tr'(R)$ is no longer relevant.*

Theorem 4.3 *Let R be a set of filtering rules and let $Tr'(R)$ be the resulting filtering rules obtained by applying Algorithm 4 to R. Then $Tr'(R)$ is free from both shadowing and redundancy.*

For space limitation reasons, we move the correctness proofs of this section to Appendix A.

5 Conclusions

In this paper we presented and audit process based on the existence of relationships between the condition attributes of the filtering rules, such as coincidence, disjunction, and inclusion. Our proposal uses a transformation process which derives from an initial set of rules to an equivalent one completely free of misconfiguration.

Furthermore, our proposal verify that any rule considered as useless is removed from the configuration. In turn, it provides an evidence of error to the security officer – he can check whether the security policy is consistent, to verify the correctness of the whole process. The complete independence between rules, moreover, enables the possibility to perform a second rewriting of rules in a positive or negative manner. If done, the security officer will have a clear view of the accepted traffic – when positive – or the rejected traffic – when negative.

Regarding the increase of number of filtering rules, because of the rewriting process, it is only significant whether the associated firewall parsing algorithm would depend on the number of rules. 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 firewall 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 [13].

As future work we are considering to extend our proposal to a more complex network setup. This paper is based on the hypothesis that only one firewall polices the network. More investigation has to be done when this role is assigned several network security components, that is, a distributed access control. Indeed, in particular, redundancy will not systematically be considered as an error [2]. It may be suited to avoid inconsistent decisions between firewalls used in the same security architecture to control the access to different zones. In parallel, we are also considering to study the anomaly problems of security rules in the case where the security architecture includes firewalls as well as IDS (*Intrusion Detection Systems*). The objective is to avoid redundant or shadowed filtering or/and alerting rules. These two works are still in progress.

Acknowledgements – 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] H. Adiseshu, S. Suri, and G. Parulkar. Detecting and Resolving Packet Filter Conflicts. In *19th Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1203–1212, 2000.

[2] E. S. Al-Shaer and H. H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *23rd IEEE Computer and Communications Societies Annual Joint Conference*, 2004.

[3] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pp. 17–31, 1999.

[4] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miege. A formal approach to specify and deploy a network security policy. In *Second Workshop on Formal Aspects in Security and Trust*, pp. 203–218, 2004.

[5] D. Eppstein, and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *12th annual ACM-SIAM symposium on Discrete Algorithms*, pp. 827–835, 2001.

[6] M. Frantzen, F. Kerschbaum, E. Schultz, and S. Fahmy. A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals. *Computers and Security*, 20(3):263–270, 2001.

[7] P. Gupta. *Algorithms for Routing Lookups and Packet Classification*. PhD Thesis, Computer Science Dept., Stanford University, 2000.

[8] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, pp. 120–129, 1997.

[9] S. Kamara, et al. Analysis of vulnerabilities in internet firewalls. *Computers and Security*, 22(3):214–232, 2003.

[10] V. Kurland, et al. Firewall Builder. *White Paper*, 2003.

[11] A. X. Liu and M. G. Gouda. Diverse firewall design. In *International Conference on Dependable Systems and Networks*, pp. 595–604, 2004.

[12] A. J. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, pp. 177–187, 2000.

[13] O. Paul, M. Laurent, and S. Gombault, A full bandwidth ATM Firewall. In *6th European Symposium on Research in Computer Security*, pp. 206–221, 2000.

[14] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *Computer ACM SIGCOMM Communication Review*, pp. 135–146, 1999.

A Correctness Proofs

Proof of Lemma 3.2 Let us assume that:

$$R_i[condition] = A_1 \wedge A_2 \wedge ... \wedge A_p, \text{ and}$$
$$R_j[condition] = B_1 \wedge B_2 \wedge ... \wedge B_p.$$

If $(A_1 \cap B_1) = \emptyset$ or $(A_2 \cap B_2) = \emptyset$ or ... 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:

$$(A_1 \cap B_1) \neq \emptyset \text{ and } (A_2 \cap B_2) \neq \emptyset \text{ and ...}$$
$$\text{and } (A_p \cap B_p) \neq \emptyset.$$

If we apply filtering 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 rule R_i applies first). Therefore, notice that $R_j[condition] - R_i[condition]$ is equivalent to:

$$(B_1 - A_1) \wedge B_2 \wedge ... \wedge B_p \text{ or}$$
$$(A_1 \cap B_1) \wedge (B_2 - A_2) \wedge ... \wedge B_p \text{ or}$$
$$(A_1 \cap B_1) \wedge (A_2 \cap B_2) \wedge (B_3 - A_3) \wedge ... \wedge B_p \text{ or}$$
$$...$$
$$(A_1 \cap B_1) \wedge ... \wedge (A_{p-1} \cap B_{p-1}) \wedge (B_p - A_p)$$

which corresponds to condition of rule $R'_j = exclusion(R_j, R_i)$. This way, if rule 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 rule 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 2 $R'_j[decision]$ becomes $R_j[decision]$, this enables to conclude that $\{R_i, R_j\}$ is equivalent to $\{R_i, R'_j\}$. □

Proof of Theorem 3.3 Notice that if R is a set of filtering rules, then $Tr(R)$ is obtained by recursively applying transformation $exclusion(R_j, R_i)$ when rule R_i comes before rule R_j, which preserves the equivalence at each step of the transformation, previously proved for Lemma 3.2. □

Proof of Lemma 3.4 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. □

Proof of Theorem 3.5 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 3.4, it is possible to commute rules R'_i and R'_j in $Tr(R)$ without changing the final decision. □

Proof of Theorem 3.6 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 $R_i[condition]$. Hence, this rule is not shadowed.

Similarly, 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 the filtering decision will change if we remove rule R_i from $Tr(R)$. □

Proof of Theorem 4.1 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 R by applying $exclusion(R_j, R_i)$ (cf. Algorithm 2) to some rules R_j in R, it is straightforward, from Lemma 3.2, to conclude that $Tr'_1(R)$ is equivalent to R.

Hence, let us now move to the second phase of Algorithm 4. 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 final filtering decision. 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 $exclusion(R_j, R_i)$ to some rules R_j which also preserves equivalence. Thus, in both cases, $Tr'(R)$ is equivalent to $Tr'_1(R)$ which, in turn, is equivalent to R. □

Proof of Theorem 4.2 and Theorem 4.3 As stated out in the proof of both Theorem 3.5 and Theorem 3.6, once shadowed and redundant rules have been removed, every rule R_i is replaced by $exclusion(R_i, R_j)$ where $j < i$.

Therefore, a similar reasoning enables to prove both Theorem 4.2 and Theorem 4.3. □