FANSA - a Framework for Automated Network Security Auditing

Ranganai Chaparadza
Fraunhofer FOKUS Institute for Open Communication Systems
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
0049 30 34637102

Mihail Peter
Fraunhofer FOKUS Institute for Open Communication Systems
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
0049 30 34637242

Ina Schieferdecker
Fraunhofer FOKUS Institute for Open Communication Systems
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
0049 30 34637241

{chaparadza | peter | schieferdecker }@fokus.fraunhofer.de

ABSTRACT

Achieving automated network security auditing in a multi-firewall enterprise network is a challenge, especially if the auditing is meant to be a continuous process that should be seen as part of the operational functions of the network. Security auditing is the practice of *evaluating* the security of systems and networks. In this paper, we present a framework for automated network security auditing. The framework is limited to the problem of designing and deploying a network security auditing system that continuously verifies that the security policy enforcements points (i.e. firewalls) installed in an enterprise network continue to function as required. We point out the problems to be addressed in this proposed framework and present some solutions/algorithms to the problems. We also present our *NetSecAuditor* tool that we developed, following this proposed framework.

Keywords

Automated Network Security Auditing, firewall rules, multi-firewall environments.

1. INTRODUCTION

Quality of Protection (QoP) of an enterprise network and some of its critical resources is *dynamic*, in the sense that the security policy enforcement points (i.e. firewalls) installed in the network and its periphery, often suffer some failures due to an overloaded state on the firewall(s) or as device failure(s) [9][10]. The other reason why QoP is dynamic is that new security policies added to some firewalls during the operation of the network may result in policy conflicts [1][2] that degrade QoP. The other factor affecting QoP is configuration errors [4], which may be introduced during the re-configurations of some firewalls. Due to these factors that affect QoP, the perceptual metric: *degree/level of protection* of a network or a resource(s) decreases accordingly. Network administrators need to ensure that QoP is kept high all the time and need to know if policy conflicts exist, whether each firewall rule works as required and whether any changes in firewall/network configurations or the dynamics of network load results in degradation of network security. Verifying all this requires automated tools for policy conflict detection and for network security auditing. The implementer of an automated security auditing system is faced with questions like: *What sort of network information is required during the auditing process?*, *How do I design and deploy such a system?*, *How do I implement an automated network security auditing/testing system that can be used to test firewalls and can keep pace with changes introduced to firewall rule specifications and to the way traffic is routed between sub-networks of an enterprise?*, A*re there any problems in firewall rule sets, that introduce problems to the testing of those rules by an automated security auditing system?*, *How about automatic derivation of test cases from firewall rules?*. The framework and our *NetSecAuditor* tool presented in this paper provide answers to such questions. **Section 2** presents the framework. **Section 3** presents the *NetSecAuditor* tool we implemented, following the framework. **Section 4** gives a conclusion and an insight into further work in this research maneuver.

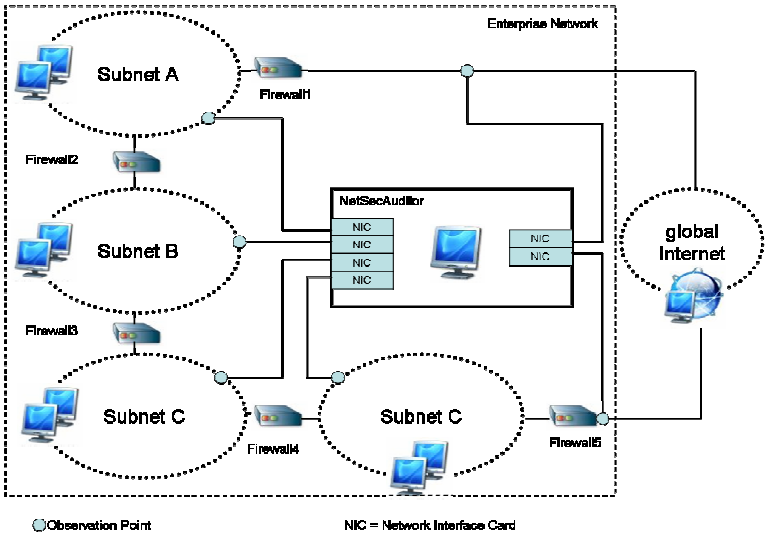

Figure 1: Security Auditing in a Multi-Firewall Enterprise Network

2. THE FRAMEWORK

The framework we present here is focussed at the problem of designing and deploying a network security auditing system that continuously verifies that the security policy enforcements points (i.e. firewalls) installed in an enterprise network continue to function as required. In particular, we target security policies

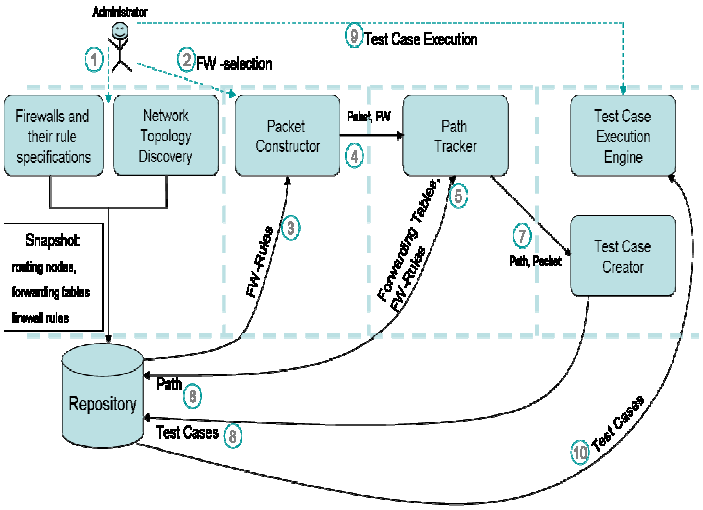

Figure 2: Interaction and *Data Flow* between the components

specified on the level of rules applied to traffic flow in and out of distributed firewalls. We propose to approach this problem by splitting the security auditing system into cooperating functional components described below. This approach makes problem analysis easier. **Figure 2** shows the components, namely the *TestPacket Constructor*, the *Path Tracker*, the *Test Case Creator*, *Test Case Execution Engine* and a *repository* for storing information required for automated security auditing (e.g. the network topology, firewalls and their rule specifications, generated testcases, test results, etc). The repository can be realized as a data base. The interaction of the components and the data passed between components are also shown. **Figure 3** shows an Entity-Relationship model that models all relevant network nodes (e.g. routers and firewalls) with relationships to their physical links, to the switches they are connected to and also, information like IP addresses, routing entries and firewall rule sets if applicable. Also a path an injected test packet would follow is modelled by putting together several HOP-information like the gateway, its inbound and outbound links and the switch the packet would pass through.

Network topology information (can be obtained using topology discovery tools [3]), routing tables, forwarding tables, information about firewalls and their rule specifications are needed in order to determine the path taken by a packet across the network given an ingress point and, to determine the actions performed on packets of particular characteristics by firewalls. By knowing the path a certain packet will take in the network, we can inject and try to detect a test packet at an appropriately selected observation point(s) along the path during the security auditing process.

Precisely, the following information about the network is required by an automated security auditing system:

- *Routing nodes*
  A fully qualified domain name, as a node's unique identifier and ip-address/interface bindings of all routing nodes inside the network, including their network interface card's (NIC) identifiers and the identifiers of switches/hubs they are connected to e.g. *router.net.com%eth0* for network interface *eth0*.
- *Routing and Forwarding tables of all routers*
  Information about the IP-address of the next HOP and the outgoing Network Interface Card.
- *Firewalls and their rule specifications*
  Firewall rules from all firewalls inside the network, ideally in the form that the Linux' firewall «iptables» [5] produces. Other simplistic firewall rule specification in the following format:
  `<source ip> <destination ip> <source port> <destination port> <protocol> <action>` is sufficient.

This information should be collected from the network by the network administrator and inserted into the repository. Because of the dynamic nature of the network e.g. changes in the routing tables etc, we consider the collected information as a snapshot in time, meaning that an automated security auditing system and the repository from which it retrieves information should always be updated with the necessary changes. Some automation tools are required to make the job of information collection and repository update easier. The challenge is to implement and maintain such tools for a number of heterogeneous systems (e.g. there are thousands of firewall and router systems out there) and information from each of the systems needs to be

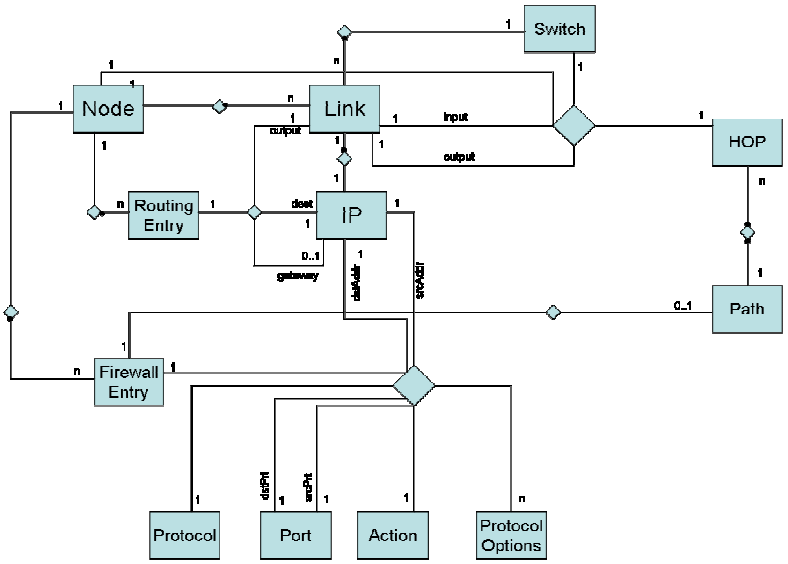

Figure 3: The information model stored in the Repository

adapted to a unified format that can be understood by an automated security auditing tool). Tapping devices, switches (via port mirroring/spanning) and hubs inside the network facilitate the observation of test packets injected by the security auditing system at different ingress points of interest in the network. The security auditing system can be deployed typically as a multi-homed machine that is physically connected to selected switches/hubs and tapping devices, such that it intercepts its self-injected test packets, destined actually to a gateway router connected to the same hub/switch (*see **Figure 1***). A number of such a security auditing system can be deployed in the network depending on the number of network interfaces (subnets) that

need to be observed in order to verify that unwanted traffic does not cross into these protected network segments due to policy conflicts and firewall failures.

Figure 2 shows the cooperation and data flow between the components. The interaction between the administrator and the security auditing system happens at *1.)* collection/wrapping network relevant information and inputting this information snapshot into the repository and *2.)* selection of a firewall for testing and invoking the security auditing task, bearing in mind that automated tools can be developed that replace the administrator, thereby gathering relevant network information, iterating over the firewalls and triggering the auditing(firewall testing) process. Refer to [11] for an example firewall testing methodology.

Table 1: Rule list example

#	Source Addr	Destin Addr	Src Port	Dst Port	Protocol	Action
A	10.0.0.1	192.168.0.0/24	21	*	TCP	ALLOW
B	10.0.0.0/16	192.168.0.1	22	*	TCP	ALLOW
C	10.0.0.0/16	192.168.0.0/24	21:22	*	TCP	DENY

3. THE NetSecAuditor TOOL AND ITS IMPLEMENTATION OF THE FRAMEWORK

We followed this proposed framework and implemented a tool called *NetSecAuditor*. In this section, we also highlight uncovered problems that need to be addressed when implementing this framework and provide solutions, developed for the *NetSecAuditor* tool. The sub-sections below describe the components of the *NetSecAuditor* tool.

3.1 The TestPacket Constructor

This module is responsible for the creation of a test packet(s) for each rule in the firewall rules specification selected for testing by a human tester (typically the administrator). A rule consists of fields of a protocol and their value specifications and, the action that is applied by the firewall (see ***Table 1***). In the automated case, a specially developed tool (replacing the administrator) iterates over the firewalls and interacts with the TestPacket Constructor. The test packet is constructed such that it is matched only by the rule under test and no other rule. The idea is to ensure that a test packet is created for each rule in the specification while avoiding field values (protocol field values) that would make the test packet matched by preceding rules that have already been tested. This is to ensure maximum test coverage for all the rules of the firewall under test. Finding the combined set of field values, covering all the required fields and being unique to a specific rule, from which field values for the test packet can be drawn, is a non-trivial task as explained below.

The input data that is needed for the creation of a test packet(s) is the identifier of the firewall under test and its complete ordered rule set (the rules specification). To retrieve this information the TestPacket Constructor queries the repository for rules of the selected firewall. The tester needs only select the firewall to be tested and pass the identifier of the firewall to the TestPacket Constructor.

The TestPacket Constructor walks the rule set sequentially and every time a new rule is picked, the module analyses the relationship of this rule to preceding rules from the same rule set, to ensure that the test packet to be generated is not matched by previously processed rules. This is done by detecting so-called *subset fields* from the previous *subset rules*. A rule *A* has a *subset field* f_iA if there is another succeeding rule *B* in the specification having the same field f_iB such that:

$$f_iA \subset f_iB$$

A rule A is a subset rule of another succeeding rule B if all fields in A are equal or are subset fields and at least one field is a subset field of rule B:

$$\forall i: fiA \subseteq fiB \wedge \exists i: fiA \subset fiB$$

For example, in ***Table 1***, field: *Source Addr* of rule A is a subset of the *Source Addr* field in rule C and rule A is a subset rule of rule C.

For every field, the TestPacket Constructor maintains a list of subset fields and calculates a union of the list elements:

$$list\_field_i = elem_1 \cup elem_2 \cup elem_3 \cup \ldots \cup elem_n$$

If this union set is equal to a field of the rule currently being processed/examined, we call this field a *covered field*. If all the fields of the rule currently being processed are covered, we call this rule a *completely covered rule*. For example, rule C from *Table 1* is completely covered by rules A and B.

Therefore, test packet creation for a *completely covered rule* is not a trivial task; we can't just pick any value from the rule's matching range, because some preceding rule(s) would match such a packet. The problem lies in finding the combined set of field values covering all the required fields and being unique to this specific rule, from which field values for the test packet targeting the completely covered rule can be drawn. Let's call this combined set of field values *Set Q*. The thing is, for a completely covered rule, there might be a combination of field values such that the resulting test packet can not be matched by preceding rules in the rules specification of the firewall under test but can *only* be matched by this completely covered rule. So, if the TestPacket Constructor detects that the rule currently being processed is completely covered by some previous rules, it runs what we call a *Coverage-Resolving Algorithm* for finding *Set Q*.

The Coverage-Resolving Algorithm

The algorithm attempts to resolve the complete coverage problem by reducing the initial range of possible values that could be used in test packet construction, to some safe set for each field of the *completely covered rule* provided the following conditions are fulfilled:

1.) There is no rule A prior to the selected rule B such that:
$\forall i,\ (0<i<=count(fields)):\ f_iA=f_iB$

2.) There is no rule A prior to the selected rule B such that:
$\forall i,\ (0<i<=count(fields)):\ f_iA \supset f_iB \vee f_iA \supseteq f_iB$

In other words, the rule set should not contain any equal or superset rules prior to the completely covered rule. Otherwise the

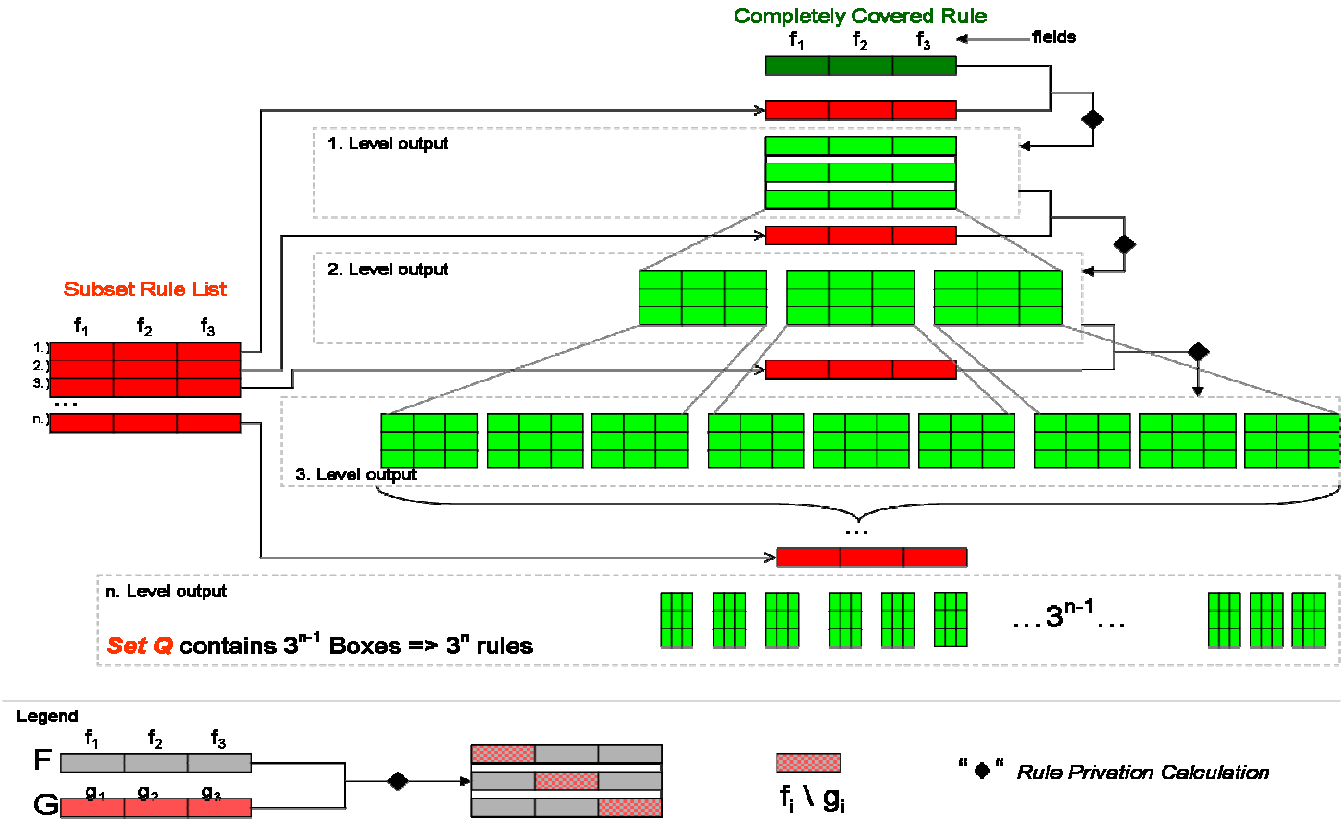

Figure 4: Illustration of the Coverage Resolving Algorithm

algorithm will fail to resolve the coverage and no test packet can be created for this rule. This is a general firewall anomaly problem and is well described in [1][2]

The algorithm works as follows: Let's consider that the current rule picked from the rule set of the firewall under test is a *completely covered rule*. Firstly, we copy all preceding subset rules of this completely covered rule into a *subset rule list {F1,F2,….Fn}*. We choose the first rule *F1* from the *subset rule list* and iterate over its fields starting from *field 1* to the last *field N* (the 5 fields in *Table 1* would result in 5 iterations). In each iteration we build *N new rules* by calculating the *privation* of the picked field while considering the current *completely covered rule* against the first rule from the *subset rule list, meaning that we find all the possible field values that are not covered by possible values of the picked field in the current iteration*. These *N rules* now form our *first level output* O_1:

O_1= {

R1=$f_{s,1}\backslash f_{1,1}$ $f_{s,2}$ $f_{s,3}$ $f_{s,4}$ $f_{s,5}$ ..$f_{s,n}$,

R2=$f_{s,1}$ $f_{s,2}\backslash f_{1,2}$ $f_{s,3}$ $f_{s,4}$ $f_{s,5}$ ..$f_{s,n}$,

…

Rn=$f_{s,1}$ $f_{s,2}$ $f_{s,3}$ $f_{s,4}$ $f_{s,5}$ .. $f_{s,n}\backslash f_{1,n}$

}

($f_{1,2}$ means field number *2* of the first rule; $f_{s,n}$ means field number "*n*" from the *completely covered rule*(the rule being processed); the operator "\" means the *privation*)

The algorithm continues with the same procedure again, but now taking the second rule *F2* from the *subset rule list* and calculating the *privation* between R1 and the second rule *F2*, iterating again over all fields inside this second rule. Let's represent this iterative operation with the sign "♦". This is repeated for every rule in the first level output (O_1), from R1 to Rn. The resulting rules from these operations built the *second level output* O_2:

$O_2 = \{R \blacklozenge F2 \mid \forall R \in O_1\}$ where ♦ represents the *iteration* as described above.

The algorithm proceeds with the third rule *F3* from the *subset rule list* against the second level output(O_2) in the same way, until it reaches the last rule in the *subset rule list*. Figure 4 illustrates this process. The output of the last level describes all possible values (the *Set Q* described above) from which a test packet, which is only matched by the *completely covered rule* and not by any of the preceding rules, can be constructed.

On **Figure 4**, we consider rules containing only 3 fields and the subset rule list containing "*n*" elements. These two variables (field number *S* and subset list size *n*) affect the complexity of the Set Q, which is a *power*: S^n. For a rule with 5 fields and more than 13 elements in its subset rule list, the algorithm produces more than 1.000.000.000 and grows exponentially. Due to the fact that we need only one possible value for each field in order to create one test packet, the implementation of this algorithm should not calculate the complete result on each output level, but only do a deep search inside the tree and stop after getting the first valid value.

It can happen, that we get an empty output set at some level and this can only occur if every privation operation for one level results in an empty set: $f_{x,y} \setminus f_{w,v} = \varnothing$. That would mean that the coverage resolution is not possible. This means the completely covered rule is skipped and processing continues with the next rule in the specification.

The output of the TestPacket Constructor includes a number of test packet descriptions, one for each rule. The packet descriptions are used by the TTCN3-Test Case Creator (described later) in order to generate real test packets and by the next module, Path Tracker in order to determine and build the packet flow path across the network. A packet description specifies concrete values for some header fields: *source ip-address, destination ip-address, sorce port, destination port, protocol type*.

3.2 The Path Tracker

This module constructs the *path* a test packet would traverse inside the network. This path is recorded and stored in the repository. The auditing system uses this information later to trigger the observation of the test packet on selected points along the path to detect the presence of the test packet. The presence of the test packet at points of interest is used in setting the verdict in a test case. The Path Tracker depends on the network topology information, the test packet description, rule sets of firewalls the test packet goes through and the starting point, where the test packet would be injected into the network. This injection point always resides on the incoming network interface of the firewall selected for testing and is passed to the TestPacket Constructor by the human tester (administrator) or by an automated tool. The Path Tracker obtains the packet description and the first firewall identifier from the TestPacket Constructor. It uses network topology information in the repository, represented by such information as switches, routers, device connectivity information and routing tables. Whenever the Path Tracker needs to construct the *path* of a packet flow it starts with the first firewall. Initially the firewall rules are analysed to determine which action the real

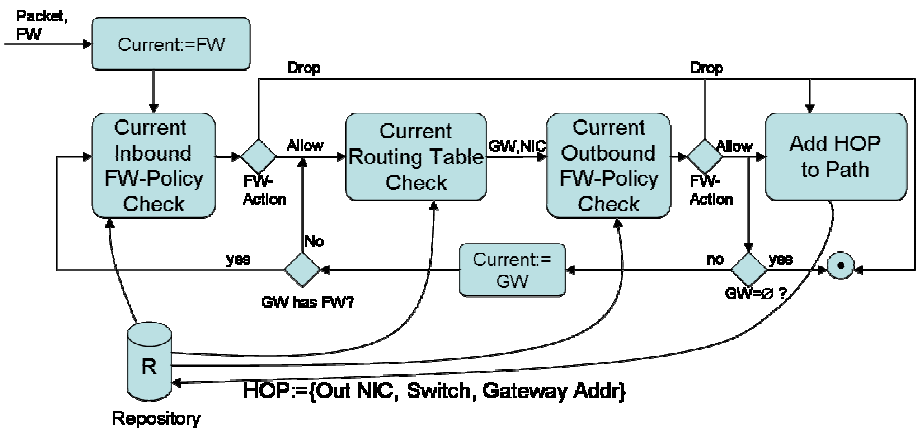

Figure 5: Path Tracker

firewall will perform upon this packet. If the firewall *accepts* the packet, its routing table in the repository is consulted and the path is determined based on the packet's destination. The routing information consists of the egress network interface and the address of the next HOP. Given the network interface identifier of the egress interface, we can identify the switch where the firewall is connected to and use the switch for observing the test packet(s). The resulting path consists of an ordered list of such triples: <*NIC id, switch id, gateway address*>. When the Path Tracker has obtained the address of the gateway used by this firewall the whole procedure starts from the beginning – it tries to retrieve the firewall rule sets from the repository for this hop (if it also acts as a firewall), check the firewall action that would be performed upon the packet, retrieve the routing table, determine how the packet is routed and add the next HOP to the *path*. This loop can be interrupted by three conditions, first – if a firewall would *drop* the packet, second – the routing information does not contain any gateway address, which would mean that the packet is destined to a machine located inside the network on the same link as the *last gateway.* And the last condition is that the repository doesn't contain any information about the next gateway. The reason for that could be a missing entry for a real existing gateway inside the network as a result of an incomplete topology discovery done by an administrator or any automated tool, but it could also mean that the test packet leaves the enterprise network at that particular network point and enters the "internet" and that's why the repository doesn't include the information about further routing of this packet – in that case the testing can be executed without any restrictions. A warning should be raised anyway, to indicate possible problems, due to we couldn't distinguish between the both possibilities. Any of these conditions would cause the algorithm to stop. **Figure 5** explains the algorithm in detail. The Path Tracker exports the *path* information into the repository and pass the packet description and the *path* to the next module – the TTCN3 Test Case Creator.

3.3 The TTCN-3 Test Case Creator (TTC-Creator)

This module is responsible for generating TTCN-3 Test Cases, which are then deployed and executed on a real test system. TTCN-3 [7] is a testing language developed by the European Telecommunications Standards Institute (ETSI) [6]. TTCN-3 provides to a tester powerful mechanisms and an easy and structured way of test development. The so-called template matching mechanisms on values of protocol message types and the support for distributed testing etc, makes TTCN-3 a powerful language for the testing of security policy enforcements points(i.e. firewalls) in networks.

After the Path Tracker has constructed the *path* the test packet would follow, the TTC-Creator uses this information to generate TTCN-3 code that controls the traffic capturing entities along the *path* taken by the test packet(s). TTC-Creator adds additional TTCN-3 code, which controls the test packet observation and the setting of test case verdicts. The test cases that produce a "fail" verdict could be made to additionally send alert messages. Also, an appropriate TTCN-3 object for a given test packet description is automatically *declared* inside the generated test case code. Further, TTC-Creator includes a packet injection routine, whose purpose is to inject a real test packet at an appropriate location inside the network.

This module uses the ETSI-developed TTCN-3 IPv6 ATS (Abstract Test Suite) for IPv6 [8] which we extended with packet type definitions for the generation of TCP, UDP and ICMP packets.

The inputs for this module are: the *test packet description* and the *path* the test packet would follow. This is passed from the previous module in the tool chain, namely the Path Tracker. The output is executable TTCN-3 test code.

3.4 The Test Case Execution Engine (TTExec-Engine)

This engine reads and executes the test case(s) selected for execution from the repository by the administrator (or automatically fetched). TTExec-Engine compiles the corresponding generated TTCN-3 test case files using a TTCN-3 compiler (*TTthree* compiler from *Testing Technologies GmbH, Berlin, Germany* [12]) and executes the test case(s). TTExec-Engine can be configured to repeat the execution of the test case(s) for continuous security auditing at a specified heartbeat.

4. CONCLUSIONS AND FURTHER WORK

Our implementation of the *NetSecAuditor* tool, following the proposed framework, shows that the proposed framework is practically implementable. We have highlighted design issues and some problems with firewall rules and have provided solutions/algorithms. In this framework, the automation of the collection of relevant network information and pushing/updating the data into repository in real-time requires some significant effort. Also, the handling of the protected subnets (that need observation) can be made difficult by the real physical network environment and complex technologies like VLANs or if one subnet is distributed over several switches (and the opposite – if a switch is serving multiple subnets), though advances in port mirroring/spanning seem to have solved the problem of monitoring multiple VLANs and multiple ports. As for firewall rules, some firewalls might include complex rules for *stateful* inspection (i.e. the case of so-called *dynamic filters*), which possibly can not be matched by only one test packet (several test packets are needed) or they might include exotic rules matching some specific message exchange(s) on a particular application protocol. For example, some firewalls do "connection tracking" as they understand the protocol semantics and perform specific actions upon the detection of some events e.g. *active ftp* requires opening a port on the client side, negotiated during the connection establishing. These issues are not covered at the moment by the framework. Also, network address translation (NAT) rules are not covered by the framework. This will be a subject for further research. Because automated network security auditing should go beyond firewall-rule testing, in the future, we seek to use lessons learned in this research to extend the framework with issues regarding automated continuous verification of Network Intrusion Detection Systems (NDIS's) also installed in the network. Future work on the *NetSecAuditor* tool will include alerting on security violations.

5. ACKNOWLEDGMENTS

Our thanks to Benoit Gaudin and Baptiste Alcalde, guest researchers at FOKUS with whom we had very constructive discussions on the *Coverage-Resolving* algorithm.

6. REFERENCES

[1] E. Al-Shaer, H. Hamed, Boutaba, Hasan: "Conflict Classification and Analysis of Distributed Firewall Policies" :IEEE Journal on selected areas in communications, vol. 23, No. 10, October 2005

[2] H. Hamed, E. Al-Shaer: Taxonomy of Conflicts in Network Security Policies: IEEE Communications magazine – March 2006, vol. 44, No. 3

[3] Y.Breitbart, M. Garofalakis etal: "Topology Discovery in Heterogeneous IP Networks: The *NetInventory* System": IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 12, NO. 3, JUNE 2004

[4] Avishai Wool: "A Quantitative Study of Firewall Configuration Errors": published by the IEEE Computer Society: 2004. http://www.eng.tau.ac.il/~yash/computer2004.pdf

[5] Linux IPTABLES/NETFILTER Firewall http://www.netfilter.org/

[6] European Telecommunications Standards Institute (ETSI) http://www.etsi.org/

[7] The TTCN3 Standard: http://www.ttcn-3.org/

[8] Work done by ETSI - Specialist Task Force: STF276

[9] The Business Case for High Availability (HA): A CyberGuard Corporation White Paper, August 2002

[10] Dr. V.C. Jones, PE: Configuration for Transparently Redundant Firewalls – White paper from Networking Unlimited, Inc May 2001 http://www.networkingunlimited.com/white001.html

[11] Firewall Testing Methodology using Spirent Solutions, July 2003: http://www.spirentcom.com/documents/1095.pdf

[12] Testing Technologies' TTworkbench: www.testingtech.de