A Unit Testing Framework for Network Configurations*

Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche

telecommunications, *networks* & *security* Research Group
Department of Computer Science, University of Fribourg
Chemin du Musée 3, 1700 Fribourg, Switzerland

Abstract. We present in this paper a unit testing framework for network configurations which verifies that the configuration meets prior defined requirements of the networks behavior. This framework increases the trust in the correctness, security and reliability of a networks configuration. Our testing framework is based on a behavioral simulation approach as it is used in hardware design [1]. The unit testing framework is part of the SNSF VeriNeC project [2].

1 Introduction

The difficulty of administration of networks increases with a network's size and heterogeneity. The VeriNeC project offers a network configuration solution, which tackles and solves these problems. VeriNeC offers an implementation-independent solution for configuring nodes in a network using XML as the configuration language. Using an implementation-independent configuration facility simplifies the task of configuration and solves the problem of heterogeneity. There is a single configuration facility the administrator will use. VeriNeC can handle the configuration of the different nodes and services in the network from the abstract XML configuration. Therefore it transforms an abstract XML configuration into an actual configuration and deploys it over a given configuration facility such as SNMP, SSH, SCP or any other.

VeriNeC offers a centralized configuration facility integrated into a graphical user interface, which helps to reduce the problem of handling vast networks. But there is more to VeriNeC: Large network configurations suffer easily from misconfiguration, causing the network to behave in an undesired way. A simulator facilitates testing existing applications in the configured network and helps understanding the behavior resulting from a given configuration. Misconfiguration can also lead to security holes in a network. Such security holes comprise unnecessary availability of services, unnecessary open firewall ports, unnecessary or wrong entries in a routing table, transport of security relevant data over unencrypted and insecure channels, and the use of empty or weak passwords.

This paper presents a tool for testing automatically a network's configuration with respect to a given set of tests; i.e. a unit testing framework for network configuration, which helps to increase trust in the correctness, security and reliability of a network.

* Supported by the *Swiss National Science Foundation* under grant number 200021-100645/1.

Jungo D., Buchmann D. and Ultes-Nitsche U. (2005).
A Unit Testing Framework for Network Configurations.
In *Proceedings of the 3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 77-82
DOI: 10.5220/0002573500770082

2 Network Configuration

Network nodes and the services they provide are configured using an XML syntax. As we want to focus on functionality of services and not specific products, the XML schema is an abstract representation of actual implementations, representing all kinds of features found in concrete configurations of given service implementations. To configure actual devices, the *Translator* module is used to generate implementation specific configurations.

A node configuration consists of a section for its hardware setup and a list of services it runs. Services are for example a server for the domain name system (DNS) or a chain-ruled packet-filtering firewall. We aimed at identifying the important features of each service and at expressing them conceptually in the schemas. Using VeriNeC, one does not configure a bind8 DNS server, but just the information needed to automatically generate the necessary configuration files for various target platforms.

2.1 Applying the Configuration

As the aim of VeriNeC is finally to configure a real network, the abstract XML network configuration needs to be distributed, applying it to actual machines.

Applying the configuration is divided into two parts. The first is the configuration language/grammar which is used natively to configure the node. This can be a Unix-style configuration file, an ini file, a batch file for an SNMP client, a batch file for Windows Management Instrumentation (WMI), a bunch of registry settings or any other kind of configuration syntax used in the actual configuration context. The second part considers the protocol with which the configuration file is distributed to the target machine. Such protocols can be simple file transfer protocols such as (s)ftp or smb to transfer plain-text configuration files, ssh, rsh and rexec to execute remote commands, SNMP to transmit the configurations, or any other existing configuration protocol. Splitting the configuration process into translating the XML to device specific configuration data and deploying that data to the machines makes that process modular, highly flexible and extensible.

The configuration schema defines the data structure for deployment. It contains commands to execute before and after the configuration data is applied, information on the deployment method, and the device specific configuration data to deploy. Configuration data can either be the content in a plain text file or arbitrary[1] XML.

The actual translation from abstract XML configuration files to configuration documents uses the XML style sheets transformation [3] language. There is one *translator* document for each implementation of each service. XSLT documents for new services or new implementations of an existing service can easily be added to extend VeriNeC.

Translation is the point where we may encounter implementation specific problems. Some implementations of a service may not support all features that can be expressed in the abstract XML configuration. VeriNeC tells the user whether he/she will lose configuration details using service implementation specific *restrictors*. The restrictors

[1] Of course, the deployment implementation must know what to do with this XML, so it better won't really be arbitrary

are implemented as XSLT documents, generating warnings for problematic tags in the abstract configuration.

The deployment is implemented in Java. A *distributor* pushes the configuration data generated by the translation process to the target system.

The translators generate device specific data, the distributor is however very generic. This allows to keep the number of deployment mechanisms small. We can use one for file distribution using various file transfer protocols and others for the configuration protocols SNMP and WMI.

3 Simulation

The network simulator [4] was initially developed to test the behavior of a configuration. The simulator is a partial rebuild of the Internet layer model [5]. We have chosen the discrete event-based simulation framework *desmoj* [6] to model the network, because *desmoj* allows to combine the event-based approach with the process-oriented simulation approach. We have implemented stateless protocols as event driven-modules and stateful protocols as process-driven modules. The simulator consists of a set of network nodes and services that can be configured using the VeriNeC XML configuration format.

3.1 The Simulator's Architecture

The network simulator is a partial rebuild of the Internet layer model with some of its protocols. As interface to existing applications we have chosen Java sockets with extended functionality. This makes it possible to use any existing application using Java sockets together with the simulator. During the simulation we replace the default factory by a factory that creates Socket implementations which redirect the whole traffic to the simulator. Because the Java sockets are constructed by means of the *SocketImplFactory*, any Java application using Java sockets redirects its traffic via the simulator sockets to the simulator, allowing the use of any socket application within the simulator. This helps us avoiding implementing these applications ourselves and allows testing any application on the network with the simulator.

The simulator offers the possibility to test whether any given application works within the configured network as expected. A graphical user interface [7] displays the simulator's log file by means of graphical animation, which helps to understand why a configuration part behaves in a certain way.

The network simulator needs events that start a simulation. Such input events can be any of the schedulable events in the simulator. Sending, receiving, dropping, and routing a packet, launching, or finishing an application are some of the existing events that can initially be fed into the simulator. Interesting are events that cause other events.[2]

Initially the simulator schedules the input events within the used simulation framework, which are then fired on the scheduled time. Normally each event causes other

[2] E.g. sending an IP Packet causes creating layer 2 events by sending the IP Packet over a layer 2 protocol, receiving the IP Packet, and passing its data to a higher-level protocol, causing layer 4 events as well.

```
<events>
 <event time="0" node="client" layer="5" service="application"
 src="client" dst="webserver" id="unique1">
  <application program="wget" parameters="http://webserver/index.html"
  type="launch"/>
 <event time="1" node="client" layer="5" service="application" id="unique1">
  <application program="dns" type="lookup" parameters="webserver"/>
  <reason configid="dns00001" />
 <event time="2" node="client" layer="4" service="udp" id="unique1"
  packetid="dns1" src="134.21.3.8" dst="134.21.6.8">
   <udp type="packetsend" srcport="45401" dstport="53"/>
</events>
```

Fig. 1. A sample output file of a HTTP client causing a DNS lookup

events to happen. Figure 1 shows the result of the processed input event. It shows a tree structure of events with the initial input event as the root. A child event is an event that is caused by its parent. Each event gives hints about which configuration part caused the node or service to behave the way it did. This can be a single configuration rule or a set of rules. These hints help the user to debug the configuration with respect to the observed (simulated) protocol run initiated by the root event.

The simulator output is hardly human readable and comprehensible. Therefore an animated graphical user interface displays all events of interest.

4 Verification

The Simulator with its GUI helps to understand the effects of a network configuration and helps to find out why the configuration may not work as desired. It is, however, not suitable to verify automatically whether a configuration satisfies a given set of properties. Therefore a unit testing framework for the network configurations was chosen to carry out the task of testing the network configurations. The unit testing framework for the network configurations uses the services of the earlier described network simulator to test the behavior against a given list of rules with desired behavior.

4.1 A Low-level Verification Language

Test cases are generally structured in five phases: Before any simulation is run, checks can be executed on each node that concern the configuration itself (so-called static analysis) and not its impact on the behavior. A candidate for such a test is the test for weak passwords. The password attributes are chosen by an *xpath* expression. Within the `configtest` element there exist a couple of possible tests like the password test, which are implemented in Java.

In the second phase a series of tests is executed which should test, whether the simulation could reach expected goals. For example, we test whether the HTTP daemon is running on the server node. These tests are within the `pretest` elements. Figure 2 presents a test whether or not the HTTP daemon is running on port 80 on the destination host. If we did not test this and at a later stage a simulation failed because no HTTP is running on the server node, we would have no information about why the test failed.

The next phase is the setup phase, where the whole testing environment is set up and the input events are scheduled. This is followed by execution of the test in the testing environment against a given set of properties. As a last step, the testing environments is deconstructed.

Figure 2 shows a network configuration test example. A test consist of a configuration which is referenced by the configuration attribute in the testcase element. The input event element contains a list of input events which are fed into the simulator. The simulation's resulting log can then be explored by a series of tests, which consist of *xpath* expressions and a target value, which the *xpath* expression should evaluate to in order for the test to pass. On failing tests, the output from the simulator gives hints about which configuration part caused the node or service to behave such that the test failed. Tests need either be grouped by the `not`, `and` or the `or`-operator. Existing test expressions are: `assertequals`, `assertcontains` and `assertexists`.

```
<testcase name="http">
  <configtests>
  <weakpassword node="webserver"
  password="/node/services/authentication/user/@password" />
  </configtests>
  <pretest>
    <and><assertequals node="webserver" expression="/node/services/httpd/@port"
      expectedresult="80" /></and>
  </pretest>
  <inputevents configuration="my_net_config"><events>...</events></inputevents>
  <test>
    <or><assertexist expression="//event[@src='client' and @dst='webserver']
      /application[@program='wget' and @type='success']"></or>
  </test>
</testcase>
```

Fig. 2. An example that ensures the availability of the HTTP service from a given node

The pretest concerns tests that are executed prior to the simulation tests. Pretests cover tests directly related to the static configuration information. Candidates for these tests are checking for weak passwords or testing whether a required service for a tests is configured and running. Within a test case, there exist two special nodes, named *example.com* and *eve.com*, which both represent nodes on the Internet, but which do not exist in the network configuration itself. On *example.com*, a number of services are running. Its purpose is testing the reachability of these services on the Internet. *example.com* is connected to the network *Internet*. Each node directly connected to this network should reach *example.com*. The other special node, *eve.com*, which represents the evil (wo)man-in-the-middle, sniffing every unencrypted (or weakly encrypted[3]) packet passing the Internet, is also directly connected to the Internet. *eve.com* is directly connected to each unencrypted (or weakly encrypted) wireless LAN and listens passively to each packet which passes such a network. In the network testing unit, *eve.com* helps finding configuration problems where confidential data can be read by means of sniff-

[3] Weakly encrypted means that the encryption can be broken easily and the transmitted messages are not properly protected.

ing. *eve.com* can also try to attack the network actively from the Internet or through wireless connections, by trying to connect to any service in the network and sending faked or invalid packets.

5 Conclusion

The immense complexity of network configuration makes using a formal verification mechanism difficult. The complexity of formal network verification results from the complex functionality of nodes and services in a network, the amount of configuration information and its impact on network behavior, and the vast size of networks and configuration files.

To cater for the practical impossibility to formally verify the network behaviour fully formally, we have developed a test framework based on a network simulator, which simulates the behavior of a network with respect to a given configuration. The correct behavior of the configured network is defined as a set of constraints that must be met by the network configuration. A constraint can be, for instance, the availability of a given service from every node inside a company's private network, which must be invisible to the outside. The set of possible constraints depends on the network and the used underlying simulation techniques. Possible future extensions to our approach are constraints on data throughput — network throughput has been neglected up to now in the VeriNeC context for the sake of focussing on functional network behavior (e.g. correct routing, packet filtering, etc.).

References

1. Haque, K., Michelson: The Art of Verification With Vera. Verification Central (2001)
2. Ultes-Nitsche, U., Jungo, D., Buchmann, D.: Verified network configuration. Technical report, University of Fribourg, http://diuf.unifr.ch/tns/projects/verinec/ (2004–2005)
3. Clark, J.: Xsl transformations (xslt). Technical report, W3C, http://www.w3.org/TR/xslt (1999)
4. Jungo, D., Buchmann, D., Ultes-Nitsche, U.: The role of simulation in a network configuration engineering approach. In: ICICT 2004, Multimedia Services and Underlying Network Infrastructure, Cairo, Egypt, Information Technology Institute (2004)
5. Tanenbaum, A.S.: Computer Networks. 4th edn. Prentice Hall (2003)
6. Bernd Page, T.L., Claassen, S.: Objektorientierte Simulation in Java mit dem Framework DESMO-J. BoD GmbH, (Norderstedt)
7. Loeffel, R.: Verinec studio, ein gui fuer den verinec simulator. Bachelor thesis, University of Fribourg (2004)