Validator: Tcl 2003

Table of Contents

Validator: Tcl 2003

Table of Contents

INFL: file3

Validator: An Agent−based firewall validation application

Clif Flynt

Noumena Corporation

`clif@cflynt.com`

Abstract

Validator is an application suite for validating firewall (and other network) behavior empiricly by generating controlled interactions on one channel while observing the behavior of the system via other channels. The application uses a suite of three sets of programs, a top−level control application which iterates through a sequence of interactions, an assistant application which coordinate subordinate applications, and child applications which interact with or monitor the behavior of the firewall.

This paper describes the architecture of the system, the flow of data and control between the three sets of programs, the format of data files, and implementation techniques used to reduce redundant code and data.

Introduction

A properly configured firewall is essential for any computer connected to the outside world. Connecting a system to the Internet without a firewall is like leaving your front door unlocked and wide open.

Unfortunately, there are few techniques for verifying that a firewall is working correctly. Current techniques for firewall validation consist of vulnerability testing with tools like SATAN, SAINT and port sniffers or intrusion detection with tools like SNORT.

The vulnerability testers will check to see what services are enabled or blocked, and perform some version checking, but cannot verify that a service has been disabled deliberately or accidently. If the service is enabled, these tools cannot determine whether the service is being provided by the firewall or a DMZ machine.

Intrusion detection is almost a requirement for a system connected to the Internet, but by the time an intrusion is detected it may be too late. By the time a file−system monitor like Tripwire is triggered, an attacker has already compromised the system. Tools like SNORT can be used to detect attacks and reconfigure a firewall at run time, but there is no guarantee that a firewall is configured to implement the specified policies correctly.

A system that initiates coordinated probes and monitors the behavior of the firewall can verify that the firewall has been configured correctly, that bad packets are rejected and logged correctly and good packets are routed to the proper hosts.

The goal of the Validator application is to fully exercise a firewall router, or set of network devices in a safe sandbox before exposing it to the hard realities of life on the internet. It does this by initiating series of interactions while monitoring the state of the system under test.

The model that Validator uses is to probe the firewall system from the outside Network Interface Connection (NIC) while monitoring the firewall behavior using the inside NIC, and by probing the inside NIC while monitoring packets that leave the outside NIC. The probes may be packet attacks, using synthetic packets that contain known exploits, or interactions with server facilities like FTP and Sendmail. The monitoring function consists of a combination of watching log files, examining the file system, and packet sniffing.

Generalizing Firewall tests

A firewall may perform many functions including filtering out undesired IP packets, routing acceptable packets to the appropriate services and internal systems, and performing NAT activities.

Here are a few examples of Validator tests.

- Spoofed Address

  An ICMP Echo Request packet is synthesized as coming from an address inside the network (say, 10.3.2.2). This packet is sent to the outside NIC. The firewall rules should reject this packet as a spoofed address, since packets from internal addresses can not reach the outside NIC. The security log file is monitored to confirm that this packet is rejected.
- ICMP Reply

  An ICMP Echo Reply packet is synthesized as coming from an address outside the network. This packet is sent to the outside NIC. The firewall rules should reject this packet for not being associated with an ICMP Request generated by this system. The security log file is monitored to confirm that this packet is rejected.
- Anonymous FTP login

  A task attempts an anonymous login FTP session, and tries to put a file to the `pub` directory. The appropriate directory on the FTP server is watched to confirm that the file is placed in the expected location.

- External service

  An IP packet with a destination outside the firewall is sent to the internal NIC. This should be routed to the outside NIC and modified by the Network Address Translation (NAT) module. A tcpdump session observes the output from the outside NIC to confirm that the packet is modified properly.

The tests to exercise these (and other) functions can be generalized with a few interaction and monitor applications. These primitive functions support traditional firewall validation as well as application firewalls.

- Probes
  Single packet sender
  : A single packet is sent from the test machine to the firewall. This packet may be a known attack packet masquerading from an outside source.

  TCP interaction
  : A complex interaction (SMTP, Web, FTP) is performed with the firewall. This may be expected to fail or succeed, depending on the test being run.
- Monitors
  Watch a log file for a pattern.
  : The application logs into the firewall and watches a given file for a given pattern.

  Examine a filesystem for changes.
  : The applications watches the appropriate system to confirm a file or files being created, deleted, or modified.

  Execute an application and scan output for pattern
  : The application spawns a child task (tcpdump, arp, netstat, etc) and checks the output for a given pattern. This may be done on the firewall (using an SSH login) or test system.

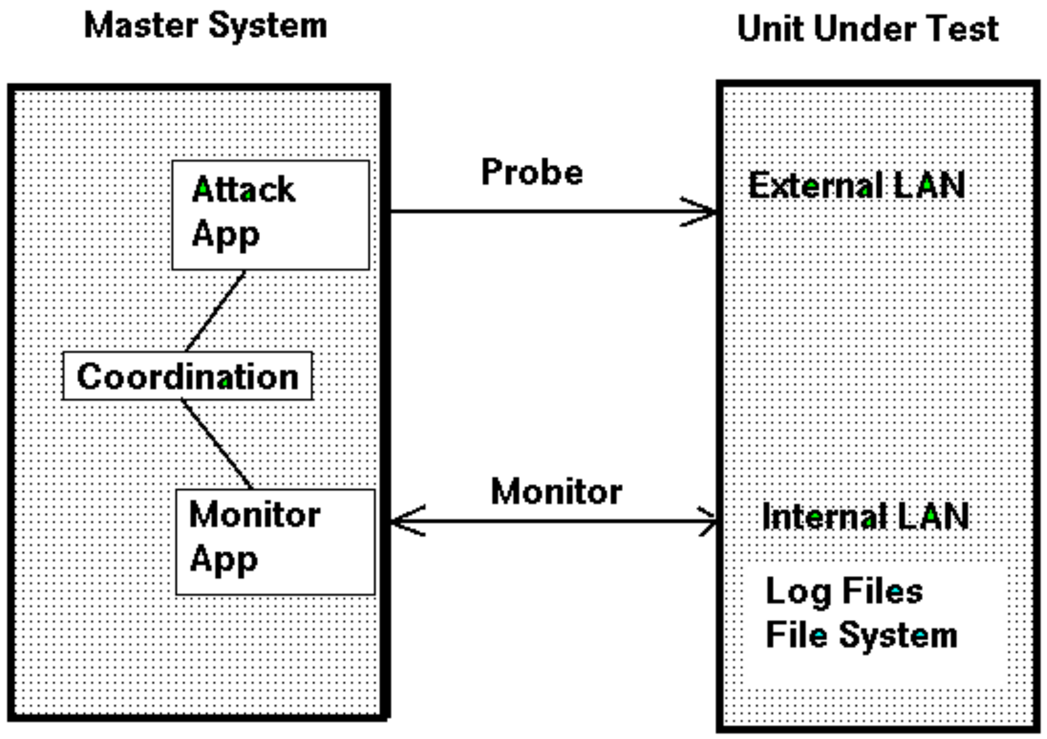
The implementation of the Validator is to use relatively small applications to perform each aspect of a test (generate a packet or monitor a log file), while other sets of software coordinate the overall test.

Hardware

The hardware configuration for Validator uses two sets of equipment: the system being characterized, and a second system that interacts with the firewall's outside interface while monitoring the system via the inside interface.

The firewall being characterized is equipped and configured as it will be deployed, with 2 or more NICs configured with the appropriate IP addresses. The deployment IP address for the outside NIC can be used even if an existing Firewall is using that IP address since this NIC is only connected to the testing system.

The testing system is also equipped with two ethernet cards, one of which is connected to outside NIC of the test system, while the other is connected to the inside NIC. The IP address of the outside NIC will be configured as necessary by the tests. The inside NIC should be configured with an *inside* IP address.



Coordinated probing and monitoring of the system under test provides complete assessment of the systems behavior

Software Architecture

The Validator architecture divides the tasks into three classes:

Test Coordinator

This task coordinates the activities of the Function Coordinator tasks to exercise a firewall system.

◊ Example: Run a sequence of tests to confirm that a firewall rejects spoofed IP packets, telnet is disabled, anonymous FTP sessions are routed to the FTP Server.

◊ This application is called the Master.

Function Coordinator

This task coordinates several Single Function Provider tasks to implement a single test.

◊ Example: Sending a packet and watching log files to confirm the packet was filtered.

◊ These tasks are called Assistants.

Single Function Provider

This task performs a single function.

◊ Examples: watching a log file for a pattern, sending a packet to a port.

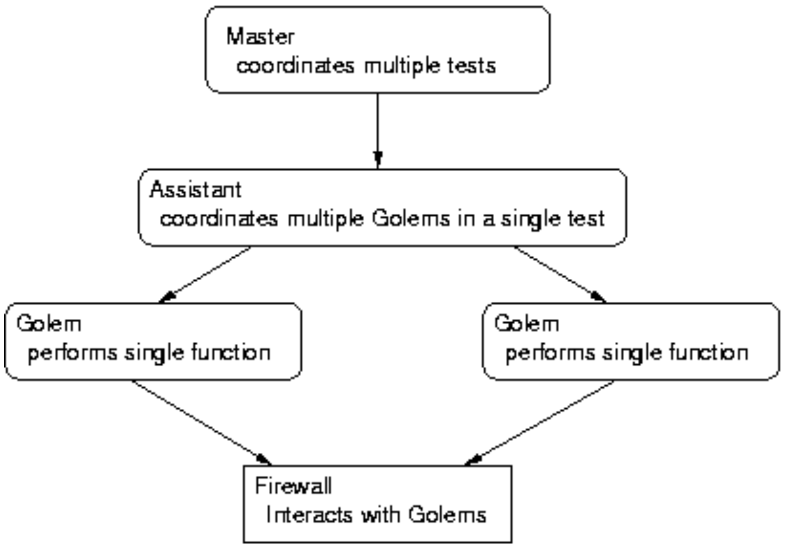
◊ These tasks are called Golems (since they are controlled by a script).

Each Golem task is started and controlled by an Assistant task, and each Assistant is started by the Master task. An Assistant may be controlling several Golems simultaenously, while the Master will only invoke one Assistant at a time.

The Object Oriented style design is not implemented within an O−O framework. The Assistant and Golems are conceptualized as virtual classes that require methods defined in the derived classes. The methods (a `runTest` procedure and the packages it requires) are merged into the virtual class at runtime.

This is implemented by using a small Tcl skeleton script that implements the common functionality for the Assistant and Golem applications (the data and methods that would be defined in a virtual class), and loading the procedures that customize the application's behavior at run time (the methods that would be defined in a derived class).

The software architecture looks like this:



The Master process spawns an Assistant for each test. The Assistant coordinates one or more Golems which perform single functions like probing or monitoring the system under test.

- Master

  The Master module reads a list of tests to run and a set of test configuration options, such as the internal and external IP and ethernet addresses of the test unit, which NIC to use for internal and external communications, etc.

  The master invokes an assistant using `exec` for each test to be run.

  The master.tcl program is a single code module that is not modified at runtime.
- Assistant

  The assistant reads configuration options from the command line. Some of the command line arguments include lists of *personality* files to be loaded at runtime to implement specific types of tests. The program flow during this test is defined by the `runTest` procedure contained in one of the personality scripts.

  The Assistant reads a rule definition file for test descriptions, and invokes one or more golems to run the tests.

  Communication between the Assistant and Golems is done via sockets. The Assistant is an TCP server, and each Golem is a TCP client.
- Golem

  The base Golem code is a simple TCP Client application. When the Golem starts, it opens a connection to a server (the Assistant that invoked it), and initiates a conversation with the Assistant.

> The Assistant sends instructions to the Golem as Tcl commands, which are evaluated within the Golem to define this Golem's functionality and control the Golem's behavior.
>
> The body of the test and data definitions are sent to the Golem from the Assistant. A Golem's initial code only defines how to connect to an Assistant and listen for commands. As the commands are evaluated, a Golem may become a probe program or a monitor.
>
> The probe scripts may generate IP packets or interact with a TCP Service on the firewall (or DMZ) to exercise that service. The monitor scripts may examine the contents of a log file, check directories for files being deposited/deleted, or examine tcpdump output to confirm that traffic does or does not occur.

The Golems are classic software agents in that they receive both configuration data and source−code scripts to be evaluated which define the actions they will perform.

When a Golem is initiated it receives an IP address and port number on the command line. The base Golem consists of a simple TCP/IP client that opens a socket to the Assistant, and waits for input. This allows golems to be run on remote sites for "in vivo" testing of a system.

The Assistant task is a TCP server. When it receives a connection from the Golem, it sends the golem the data definitions and all scripts that will be needed to perform the test.

This table shows how the application control flows from the Master to the Assistant and finally to the Golems, and provides an overview of the message flow.

Master	Assistant	Golem
Master invokes Assistant		
	Assistant invokes Golem	
		Golem opens socket to Assistant
	Assistant sends data and script to Golem	
	Assistant tells Golem to start test	
		Golem interacts with Firewall
		Golem reports result to Assistant
	Assistant collects Golem results	
	Assistant sends final result to Master	

Activity and Message Flow for a single test.

The Assistant is expected to run on the same hardware as the Master, with access to the full set of scripts. The Master uses the `exec` command to spawn an Assistant. The Assistant is customized via command line arguments that define the scripts to be sourced at runtime. The final result of a test is returned to the Master program with a `puts` command.

The Golem may run locally, or remotely. In the current implementation, the golem is always invoked on the test platform. The Golem uses an agent style design in which the base golem is a very simple TCP client that uses the Tcl "eval" command to process data received from the Assistant. The Golem receives all data values, procedures, etc via the TCP socket.

Implementation Details

Configuration Data and Files

Data duplication is reduced by partitioning the data into subsets based on the value's persistance. Data that is used throughout a systems validation, for example, the Ethernet addresses of the NICs, is maintained in an initialization file. Data that is used by a specific test type, for example the path to a log file, is maintained in a test definition file. Data that is used by a single instance of the test, for example, the pattern to scan a test for, is defined in the set of test rules.

These sets of data are read by the Master application, and propagated to Assistant to Golem as necessary. In many cases, the substitution is performed in the Master, and the Assistant and Golem receive scripts with hardcoded values to evaluate.

File Name	Description	Examples

NAME.ini	Data that is persistent across a set of tests
	
	These values are maintained in a common data space.
		`TSTinsideETH`
		Ethernet address of testsystem internal NIC
		`FWoutsideETH`
		Ethernet address of firewall outside NIC

NAME.rul	A list of tests and per test data definitions.
	
	These values are separated into specific areas for the Assistant and each Golem.
		`user`
		A user id for this test only.
		`passwd`
		A password to use for this test only.

NAME.def	Definitions related to a generic test.
	
	The defined keywords are substituted with appropriate values before being transmitted to the Assistant.
		`g-scripts`
		The scripts to be downloaded to the golem
		`g-conv`
		The conversation for an `expect` interaction.

Data definitions are read by the Master program and propagate to the Assistant and Golem applications

The `.ini` and `.rul` files are expected to be modified by a user to reflect the set of tests that needs to be done. These files consist of fairly simple keyword/value pairs.

A subset from an `ini` file resembles this:

```
FWoutsideNIC  eth0
FWoutsideETH  00:0c:29:04:d7:c3
user     clif
passwd   testing
```

The `.rul` file is slightly more complex in that it has two pieces of data for the Master program, and then sets of keyword/value pairs to be transferred to the Assistant and Golems.

The format is to use a single line for each test (newlines escaped with backslashes as necessary) and Tcl lists for sets of keyword/value pairs to be submitted to child applications.

The next exmaple is a test script that confirms that anonymous FTP is enabled. The test is named `AnonymousFTP`. It uses the `FTP-S` test pattern. There are no special arguments for the Assistant. Only one Golem is needed by this test, and it uses a user name of `anonymous` and password of `foo@bar.com`. It expects to succeed.

```
AnonymousFTP FTP-S {} {user anonymous passwd foo@bar.com checks OK}
```

The `.def` files are more complex, and will only need to be modified by someone creating new individual tests. The test definition files each contain one or more test definitions. The definitions use multiple lines, and are separated with a pair of newlines (a blank line).

Each test definition consists of one or more fields. The first field is the name for the test, and subsequent fields are Tcl formatted lists of key/value pairs for the Assistant and each Golem.

This is the test definition for the FTP pattern. The first list of arguments tells the Assistant to source the `a_simple.tcl` script (that simply starts the golems and waits for results), and defines the interface, IP address and port for the Assistant. The second list of arguments is used by the Assistant to modify the scripts sent to the Golem. This defines the user and password to use for the FTP conversation, and the script files that contain the FTP conversation code.

```
FTP-S {-i $TSToutsideNIC -dest $FWoutsideIP
    -personality a_simple.tcl -assistIP $assistIP -port $port} \
 {
   -user $user -password $passwd -checks $checks -appl ftp
   -dest $firewall
   -g-scripts {expectBase.tcl g_expectConvers.tcl }
   -g-conv {g-convers-ftp}
  }
```

The data flow when the test is evaluated resembles this:

Master	Assistant	Golem
.ini and .rul file names are on command line.		
load .ini file and assign values to a common data repository.		
load .rul file and evaluate first test rule.		
New data repositories are created for the Assitant and Golems. The data from the `.ini` file is merged into these repositories, along with data defined in the test rule. If there are collisions, the data in the test rule overrides the data in the `ini` file.		
load .def file and extract test definition.		
perform substitutions on Assistant and Golem arguments for defined values.		

assemble Assistant command line based on test definition.		
invoke Assistant with assistant and golem options in command line.		
	extract assistant state variable settings from command line.	
	load personality script files.	
	invoke golem with Golem identifier, Assistant IP address and port on command line.	
		open socket to Assistant and send identifier (this lets the assistant link the golem identifier and channel)
	extract golem state variable settings from command line	
	send scripts and variable definitions	
		evaluate test and send result
	compare individual golem returns to expected returns, return combined output	
		display test results and step to next test.

Details of Control and Data flow as a test is evaluated.

Propagation of data values

In order to be useful, the Validator must be configurable by a system administrator, not a Tcl expert. While it would be simplest from a coding point of view to make the configuration files Tcl scripts this would reduce the acceptability of the applications.

The data in the configuration files are maintained as key/value pairs, with simple grouping via curly braces. This pattern is used by enough system administration packages that most administrators will be comfortable with it.

Initially, the data was maintained in associative arrays using the configuration file keys as an indices.

The Tcl `array set` command makes it easy to import the keyword/value pairs. This set of code reads an `.ini` file and fills the `baseState` associative array:

```
  set if [open $fileName r]
  set def [read $if]
  close $if
  array set baseState $def
```

Saving the data in associative arrays provides a clean mechanism for organizing the various sets of data, but it leads to a complication when user−friendly strings like `−user $user` need to be substitituted internally as `−user $golem1(user)`

The initial solution to this was to use the `regsub` comamnd to convert between the two forms. This solution became fragile as the application moved from proof−of−concept to real−world status.

The current solution is to save the key/value pairs in a child interpreter as global variables within the interp. The rule definitions can then be substituted without modification with command like `$child eval {subst $commandString}`

The advantage of using an `interp` as a data repository over using a `namespace` is that code evaluated within a namespace will attempt to resolve variable names from the global scope if they are not defined within the namespace. Code evaluated within an `interp` will not try to resolve variables from a parent interpreter. Using the `interp` avoids unexpected promotion from namespace scope to global scope in the variable name resolution.

Avoiding code duplication

One of the implementation goals for this project was to keep the amount of duplicated code to a minimum. It was expected that as the project evolved changes would become necessary, and having multiple sets of code to update would make the project difficult to maintain and extend.

For example, while the `expect` extension is the ideal tool to use to watch for patterns in log files and packet sniffer output, an `expect` script can require a lot of redundant code. Conversations written as a series of `expect` commands can become very long, and each `expect` interaction with a child process should include `timeout` and `eof` patterns.

A solution to this is to reduce the `expect` conversations to a minimal list of patterns and actions, and allow the Golem to expand these into actual `expect` commands at run time.

For example, the `g−convers−ftp` script consists of a list of lists, the first of which is shown below:

```
{
      "Name *:" {
        exp_send "$State(user)\n"
      }
      "is unreachable" {
        set STATUS "FAIL"
        set State($id.info) "No route to host"
      }
      "tion refused" {
        set STATUS "FAIL"
        set State($id.info) "No Service"
      }
}
```

This set of lists is sent to the Golem, which reworks each conversational element into an `expect` command that includes `timeout` and `eof` patterns if these patterns aren't in the original conversation list.

```
  expect {
      "Name *:" {
        exp_send "$State(user)\n"
      }
      "is unreachable" {
        set STATUS "FAIL"
        set State($id.info) "No route to host"
      }
      "tion refused" {
        set STATUS "FAIL"
        set State($id.info) "No Service"
      }
      eof {
        set State(status) FAIL
        set State($id.info) "Child closed at $State($id.conversationNum)−$position"
      }
      timeout {
        set State(status) FAIL
        set State($id.info) "Timeout at $State($id.conversationNum)−$position"
      }
}
```

Tcl Strengths

While it's probably true that any computing project can be done in any language, some applications and languages just work well together. This application would be much more difficult in any other language.

Using Tcl and the Agent/Socket architecture allows the base code for this application to be written in under 2000 lines. This is a scope that's easy for a single developer to create and maintain. At this scale, most of the developer's effort is expended solving the problems of firewall interactions and collating results.

Using more traditional languages and designs, the application would easily have been 10 times as large, and most of the developer's time would be spent developing and maintaining the code framework, instead of developing tests and solving the actual problem of characterizing a firewall.

Tcl features that made this application possible include:

- string handling capabilities.

  This application has many sections that format text or search text for patterns. Tcl's string, regular expression, and format commands provide all the required hooks for this.
- the ease with which Tcl can be extended to perform new tasks (`expect` and packet generating extensions).

  Most of the interactions with the Firewall system are implemented using the `expect` extension. The packet generator is a SWIG generated Tcl extension written for the `libnet` library. (This is described in a separate *Tclsh Spot* article.)
- the ability to organize and compartimentalize information in separate tasks, child interpreters and associative arrays.

  Keeping the individual subsystems completely separate from each other is a key to making this application maintainable. The ability to partition the application into separate tasks, and then to partition data into leak−proof child interpreters provides most of the functionality for this.

The associative arrays provide an open ended data structure for organizing the data at run time. This data structure has an advantage over traditional linked lists or structs in that the data collections and individual fields can all be easily defined at runtime by the data, rather than being defined within the application code.

- the socket support.

  The simple elegance of the Tcl socket commands make developing a Client/Server architecture a matter of minutes instead of hours of development time.
- the ability to evaluate commands and modify an application at runtime.

  The functionality provided by the `eval` command is critical to the Agent style architecture. Trying to implement this project in Java or "C" would have required a much different (and more difficult) architecture.
- the ability to catch exceptional conditions on single commands or large code segments.

  Applications that must deal with something in the real world, instead of just algorithms, must be able to cope with unexpected conditions. This goes double for applications that are testing and exercising a possibly mis−configured system.

  The `catch` command provides the ability to evaluate a single command or a large block of code and capture the status for later processing.

  The ability to nest `catch` commands in this manner allows the application to run a large interaction without crashing, and still capture individual execptions that require unique processing.

Future Work

- Incorporate TLS

  The current implementation of the Validator runs on a single system that exists safely behind a firewall. It uses simple TCP sockets to send the root password from the Assistant to the Golems.

  In order to be marginally safe to run a Golem on an outside system, a secure socket should be used instead of a simple TCP socket.
- Use tcllib instead of expect scripts

  Expect provides a generic interface to any text oriented process.

  However, different versions of programs use slightly different prompts and error messages, which can make tuning an Expect script tricky and possibly fragile across platforms.

  The data protocols are more tightly adhered to. Tcl packages like the `ftp, smtp, http` and `pop3` packages will provide a more platform neutral implementation of these tests at the cost of slightly more code.
- Expand test definitions

  To be useful, out−of−the−box suites of test rules to validate simple Linux and BSD based firewalls will be needed. These rule sets would allow folks configuring a firewall to confirm that the rulesets they are using are actually doing what they want.

Some vendors are providing applications that translate from human−friendly firewall policy desciptions to firewall rules. The Validator application should also read these human−friendly descriptiosn to generate a set of tests to verify that the policy is actually implemented.

- More user−friendly summaries of test results

  The current application reports pass/fail conditions for each test and each golem. This is adequate for a developer familiar with the application, but too much unformatted data for someone oriented towards provisioning a firewall rather than developing tests. It will be relatively easy to add a GUI front end to the Master process to select tests to run, create `.ini` files and present digested results to the user.

Summary

The Validator application provides a small framework that supports a large range of tests to validate and verify firewall behavior. It does this by initiating some interaction on the firewall's outside port while monitoring the firewall state via the inside port. The application can also monitor other systems inside and outside the network to confirm that services are being performed on the proper DMZ machine or external network, and can watch a packet sniffer (tcpdump) for low level analysis of what packets hit the wire.

The current status is that the framework is complete and is being used for small scale testing. Improvements of the framework are continuing as more extensive tests are developed.

Tcl was chosen for this application based on its ability to perform runtime configuration of the individual applications, and support for string manipulation, controlling other applications, and easy−to−use socket support.

Conclusion

The Validator fills a void in network policy validation. Other tools do not provide the ability to both probe and monitor the system under test in a controlled manner.

Tcl is an excellent language for this application. The development time for Validator was slightly under 2 man weeks. This compares well to the 3 man weeks it took to write the paper describing the project.

Acknowledgement

Thanks to Dan Razzell for reading, commenting and helping me clarify my thoughts.