Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Automatic Regression testing of network code: User-Mode Linux and FreeSWAN

Michael C. Richardson
Sandelman Software Works Inc.
mcr@sandelman.ottawa.on.ca http://www.sandelman.ca/mcr

Abstract

The Linux FreeSWAN project (IPsec for Linux) produces rather complicated networking code. The successful application of the protocol results in all network data being encrypted. The use of dynamic keying means that it nearly impossible for an observer (even a trusted one trying to test) to know what is going on. The need for multiple systems (often as many as 6) to be properly configured creates an environment nearly impossible to test regularily.

The emergence of virtual machine technology, particularly, User Mode Linux, has provided a solution to the testing problem: create as many virtual machines as needed and control them using standard testing scaffolding technology: expect(1). This paper describes the scaffolding and the resulting testing regime which is used.

The focus is around a modified network switch emulator, "uml_netjig" which provides the ability to play and capture network packets through a single User-Mode Linux virtual machine.

A second iteration of this tool is also described, combining more complicated expect scripts, and a command mode for uml_netjig, permitting coordination of the multiple virtual machines that are needed when doing fully negotiated IPsec sessions.

1 Background: What is this about

The Linux FreeS/WAN project is a funded project. It has the mandate to produce an IPsec implementation for Linux. The ultimate goal of this effort is to provide systems and software to permit citizens for the world to keep all of their Internet traffic private.[1].

Testing networking protocols is often difficult. By definition there is at least one network involved and often several independent systems attached to the network.

With many application layer network protocols (e.g. http) one can cheat—the network is the virtual "loopback" device, and multitasking permits both ends of a protocol to run on the same host. It is therefore common to see people doing all sorts of network development using a garden-variety notebook.

The situation is not the same for transport and network layer protocols such as IP, TCP, and

[1] See http://www.freeswan.org/

IPsec. These layers of the protocol are more fundamental. They are typically implemented inside a system kernel. This makes development work as difficult as generic kernel work.

If one is to test them on one's notebook or desktop, one risks putting one's own development environment at risk. It is common experience that doing kernel development is much easier with at least two machines—one machine is crashed every ten minutes and the other machine is used as the development host. The split between development and testing is much better understood in embedded system work—the machine under test is often of a totally different type than the development machine. Historically, the machine under test (a VCR or a modem) is incapable of even running a development environment.

Network protocol development work is further complicated by the need to have more than one machine involved.

1.1 eXtreme Programming

The growing discipline of eXtreme Programming[Bec01] has a number of fundamental principles

- rapid feedback
- assume simplicitiy
- incremental change
- embracing change
- quality work

[Bec01] goes on to explain that the fundamental activities are coding, testing, listening, and designing. XP tries to reach the point where one writes the test cases before the code. To do this, the cost (in effort and time) of testing must be reduced such that all tests can be run frequently—several times a day if possible. This very rapid feedback reduces the risk of introducing problems—permitting developers to program more efficiently and with more confidence.

This paper describes the typical requirements for doing network testing. The reasons why it is expensive and why it is difficult to automate are explained. Our solution uses User-Mode-Linux to turn machines into processes. The scaffolding is then used to control these processes, to put them through their paces on a regular basis.

No solution is perfect on the first pass—XP actually encourages partial solutions to be implemented and feedback to be received—so we describe our second pass, which at the time of writing, is still in the design phase.

2 How to test with physical hardware

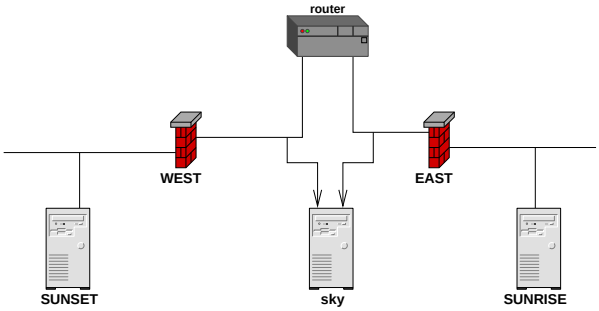

Figure 1: Basic Physical Network configuration

The basic network is shown in Figure 1. The taxonomy for our test setup is that the Sun rises

in the east and the sun sets in the west. Thus one can easily remember where each host is.

EAST and WEST, shown with firewall icons, are FreeS/WAN IPsec gateway boxes.

SUNRISE and SUNSET are just ordinary hosts whose traffic will be protected by their respective gateways.

The machine SKY is used to do network analysis ("sniff"). There are frequently problems that occur when trying examine the traffic produced by a machine itself, so a separate machine to make unbiased observations if necessary.[2]

The two gateway boxes are not directly attached, but rather are connected via a router. There are two reasons for this:

- the current implementation of FreeS/WAN requires a default route to operate correctly.
- a common operational issue is with links where the Maximum Transmission Unit (MTU) is restricted, and this router provides a place to cause such an impairment[3]

This setup is very representative of the typically deployed scenario for FreeS/WAN systems in a VPN. It does not cover every single situation—most of the most difficult-to-reproduce bugs have occured in other setups. More machines are needed to create such setups.

Aside from the space and cost involved in providing each developer with six machines (it is often the case that `sky` is the developer's desktop), there are a number of other factors that make this difficult.

The major problem is maintaining this setup. There are many machines with many files that must be maintained. The systems must be kept up-to-date so that the latest kernels can be tested, yet at the same time, testing against older kernels is necessary. Different distributions need to be tested. The combinatorics are quite high.

The other major problem is work environment. Sitting in a room with six computers is a lot of noise. Getting access to each system's console is difficult (one can not rely upon network logins!). If a monitor is attached to each system (vs a monitor switch), then the developer probably gets too much exercise.

One answer to this is serial consoles. See Figure 2. Terminals attached to serial ports was the primary way that people used Unix until the advent of the X-terminal, and Linux continues this grand tradition.

One simply puts the following in `/etc/lilo.conf`:

```
serial=0,38400n
...
image=/boot/vmlinuz-2.4.18-6mdk
   label=linux2418
   root=/dev/hda1
   append="devfs=mount \
     console=ttyS0,38400 \
     console=tty0"
   read-only
```

[2] In particular, on Linux 2.2 or lower, turning on the packet capture mechanism changes the control structures attached to the traffic and causes faults relating to policy for the keying channels' control packets. PR#48 at http://bugs.freeswan.org:81/bugs/gnatsweb.pl 2.4 has solved this problem

[3] FreeS/WAN has adopted the term "impairment" to denote any challenges which are introduced to a system or network to permit another part of the system to be tested

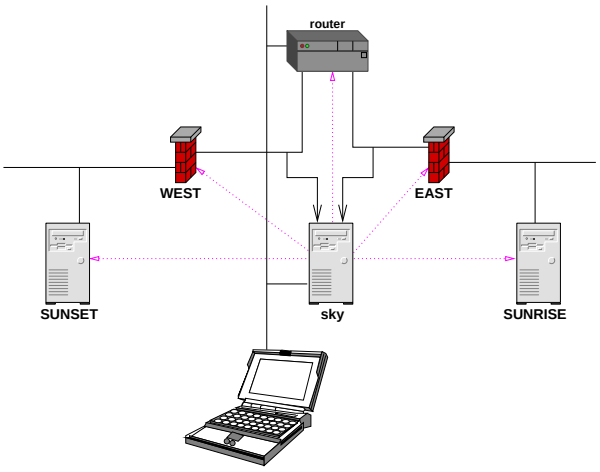

Figure 2: Basic Network with console access

The console then appears on both “COM1” and on the VGA screen. In this situation, the machines may be located in another room, connected to a console server. One logs in from one’s (quiet) desktop to the console server, accessing each machine via a serial port. Serial interfaces are readily available with either PCI or USB interfaces. This makes building a 6-port console server rather easy.

The developer now has ready access to each machine, can reboot each machine, select different kernels, and can configure it without even having networking on. In addition, kernel panics (“kernel oops”) or other strange output on the console can be cut and pasted into emails, etc..

2.1 Still challenging to test

The serial consoles do not solve the other problems—managing the very many different configurations, or coordinating the systems to perform a test case.

The author has used such a setup for many years with many Unix operating systems. Using the “expect” program and the serial consoles one can automate some of the tests. Some of tests are harder to deal with—ones that fail can cause the system to hang—this will require operator intervention. Further use of more hardware can solve this problem as well—relays can toggle reset switches or even power cycles.

The result, however, is a very complicated testing environment—it can take weeks to configure it, and mere hours to break. There is far too much specialized hardware involved, not to mention the software.

There is a better way which will be described, but first, the requirements for the testing environment will be examined in a bit more detail.

3 What do we really need

3.1 A brief primer on IPsec

IPsec[TDG98],[KA98a] consists of three transport layer protocols: AH[KA98b], ESP[KA98c] and IPcomp[DNP99]. There is one management protocol in existence at this time, ISAKMP[MSST98]/IKE[Pip98],[HC98].

These transport protocols can be applied to upper layers of TCP, UDP, or any other transport protocol. When the upper layer is the “IPIP”[Per96], then the protocol is said to be in “tunnel” mode. For most Virtual Private Network (VPN) usages, tunnel mode is the preferred method since it hides the origin source/destination address. VPNs are often treated as being virtual leased lines.

Each of the transport protocols provide session-layer encryption. They are referred to

as “security associations.” These are unidirectional concepts—a pair is usually needed for bidirectional communications.

3.1.1 Authentication Header (AH)

The Authentication Header provides origin authentication and integrity of the headers and of the data portion. No privacy is provided.

3.1.2 Encapsulating Security Payload (ESP)

The ESP header provides origin authentication, integrity and optional privacy of the data portion only. Normally, this privacy option is provided by encryption, but the specification permits a “null” encryption to be used in some circumstances.

3.1.3 IP compression header (IPcomp)

A good encryption algorithm produces cyphertext that is evenly distributed. This makes it difficult to compress. If one wishes to compress the data it must be done prior to encrypting. The IPcomp header provides for this.

One of the problems of tunnel mode is that it adds 20 bytes of IP header, plus 28 bytes of ESP overhead to each packet. This can cause large packets to be fragmented. Compressing the packet first may make it small enough to avoid this fragmentation.

3.1.4 Internet Security Association Key Management Protocol (ISAKMP)

ISAKMP is a framework for doing Security Association Key Management. It can, in theory, be used to produce session keys for many different systems, not just IPsec.

3.1.5 Internet Key Daemon (IKE)

IKE is a profile of ISAKMP that is for use by IPsec. It is often called simply “IKE.” IKE creates a private, authenticated key management channel. Using that channel, two peers can communicate, arranging for sessions keys to be generated for AH, ESP or IPcomp. The channel is used for the peers to agree on the encryption, authentication and compression algorithms that will be used. The traffic to which the policies will applied is also agreed upon.

3.2 Testing KLIPS

In FreeSWAN, the session layer encryption, security association management and traffic selection is done by a kernel component called KLIPS (Kernel Level IP Security). This component can be built as a loadable kernel module or statically built in.

As the security associations are unidirectional one can effectively separate the encrypt/encapsulate and decrypt/decapsulate operations for testing purposes.

For ease of thinking, the encryption operations are always done on EAST and the decryption operations are always done on WEST.

As indicated in Figure 3, a source of plaintext packets is needed, a way to examine the cipher-

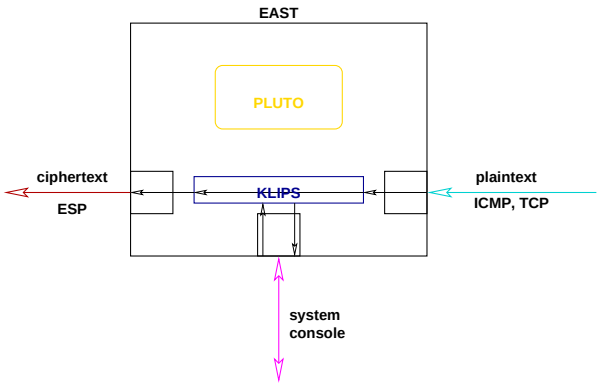

Figure 3: How to test KLIPS

text packets is needed, and a way to configure the system is needed. In the physical setup of the previous section, the source of plaintext packets is provided by the machine SUNRISE, and the examination of the packets is provided by SKY.

A typical initialization script for KLIPS is shown in Figure 4.

The term SPI means "Security Parameters Index." Each security association is indexed by a SPI. Note that a separate SPI is setup for the ESP operation and for the tunnel operation. The two are then grouped together.

The `eroute` (Extended Route) command then selects traffic by source and destination address for processing by the aforementioned group. [KA98a] defines other selectors, including TCP and UDP port numbers, but those selectors are not implemented in KLIPS at this time.

The `tncfg` command attaches the IPsec pseudo to a physical device. This is necessary in 2.0 and prior kernels to provide a path for the resulting packets to actually leave the system. Otherwise, the `route` command at the end can cause packets to loop internally. Eliminating this problem—we refer to it as "stoopid routing tricks™"—is the major goal of revisions to KLIPS.

The `ipsec klipsdebug` commands turn on various debugging output. This debugging output is important for diagnosing what has really happened when the system fails.

Finally, the `ipsec look` command produces a short summary of resulting system setup. The output of this appears in Figure 5.

At this point, the system is ready to have packets sent through it. If the packets match the criteria for the SA, then they will be encrypted with the provided key.

3.2.1 KLIPS hassles

The observant will notice a number of numbers in the above output which were not in the script: the IV field (`0x24a4a14e81ee960e`), the lifetime values (it has been 9 seconds between the SA was created and the look command occured), and the date.

These variances cause two problems: the console output is not consistent on every run, and the resulting encrypted packets will have different ciphertext on each run.

Figure 4: A typical initialization script for KLIPS

```
#!/bin/sh
TZ=GMT export TZ

ipsec spi --clear
ipsec eroute --clear

enckey=0x40434345454646494a4a4c4c4f4f515152525454575758
authkey=0x87658765876587658765876587658765

ipsec klipsdebug --set pfkey
ipsec klipsdebug --set verbose

ipsec spi --af inet --edst 192.1.2.45 --spi 0x12345678 \
  --proto esp --src 192.1.2.23 --esp 3des-md5-96 \
  --enckey $enckey --authkey $authkey

ipsec spi --af inet --edst 192.1.2.45 --spi 0x12345678 \
  --proto tun --src 192.1.2.23 --dst 192.1.2.45 --ip4

ipsec spigrp inet 192.1.2.45 0x12345678 tun inet \
  192.1.2.45 0x12345678 esp

ipsec eroute --add --eraf inet --src 192.0.2.0/24 \
  --dst 192.0.1.0/24 --said tun0x12345678@192.1.2.45

ipsec tncfg --attach --virtual ipsec0 --physical eth1
ifconfig ipsec0 inet 192.1.2.23 netmask 0xffffff00 \
  broadcast 192.1.2.255 up

# magic route command
route add -host 192.0.1.1 gw 192.1.2.45 dev ipsec0

ipsec look
```

Figure 5: Output of `ipsec look`

```
east Tue Apr  2 04:32:28 GMT 2002
192.0.2.0/24       -> 192.0.1.0/24
       => tun0x12345678@192.1.2.45 esp0x12345678@192.1.2.45  (0)
ipsec0->eth1 mtu=16260(1500)->1500
esp0x12345678@192.1.2.45 ESP_3DES_HMAC_MD5: dir=out src=192.1.2.23
iv_bits=64bits iv=0x24a4a14e81ee960e alen=128 aklen=128 eklen=192
life(c,s,h)=addtime(9,0,0)
tun0x12345678@192.1.2.45 IPIP: dir=out src=192.1.2.23 life(c,s,h)=addtime(9,0,0)
Kernel IP routing table
Destination  Gateway      Genmask         Flags   MSS Window  irtt Iface
192.0.1.1    192.1.2.45   255.255.255.255 UGH      40 0          0 ipsec0
192.1.2.0    0.0.0.0      255.255.255.0   U        40 0          0 eth1
192.1.2.0    0.0.0.0      255.255.255.0   U        40 0          0 ipsec0
192.0.1.0    192.1.2.45   255.255.255.0   UG       40 0          0 eth1
192.0.2.0    0.0.0.0      255.255.255.0   U        40 0          0 eth0
0.0.0.0      192.1.2.254  0.0.0.0         UG       40 0          0 eth1
```

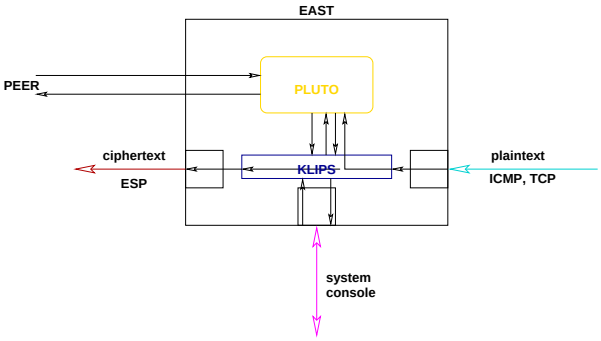

Figure 6: How to test Pluto

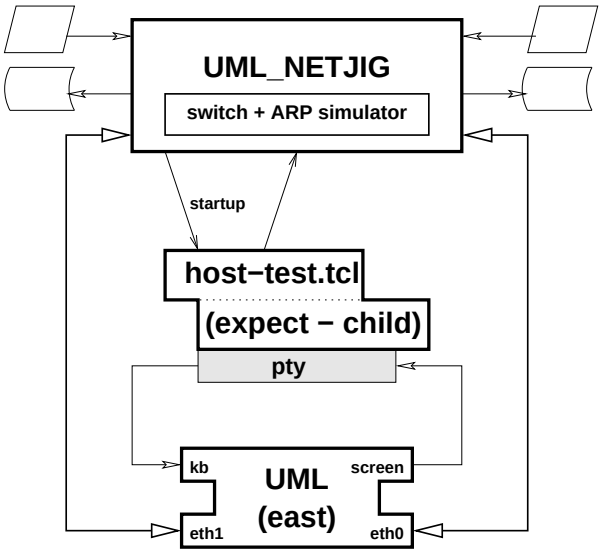

Figure 7: NetJig interface diagram

3.3 Testing Pluto

4 The first virtual attempt

4.1 How to configure to use “make check”

4.1.1 What is “make check”

“make check” is a target in the top level make-file. It takes care of running a number of unit and system tests to confirm that FreeSWAN has been compiled correctly, and that no new bugs have been introduced.

“make check” expects to be able to build User-Mode Linux kernels with FreeSWAN included. To do this it needs to have some files downloaded and extracted prior to running “make check”. This is described in the FreeSWAN documentation, un-

der http://www.freeswan.org/freeswan_snaps/CURRENT-SNAP/doc/umltesting.html.

4.2 Running “make check”

“make check” works by walking the FreeSWAN source tree invoking the “check” target at each node. At present there are tests defined only for the `klips` directory. These tests will use the UML infrastructure to test out pieces of the `klips` code.

The results of the tests can be recorded. If the environment variable REGRESSRESULTS is non-null, then the results of each test will be recorded. This is used as part of a nightly regression testing system.

“make check” otherwise prints a minimal amount of output for each test, and indicates pass/fail status of each test as they are run. Failed tests do not cause failure of the target in the form of exit codes.

5 The second virtual attempt

This attempt is illustrated in Figure 8.

6 Conclusions

Use of virtual testing environment massively simplifies automated tests.

Limitations are that one can only test Linux 2.4 and beyond kernels.

It takes a lot of RAM and a lot of CPU, but still is cheaper than coordinating many physical machines.

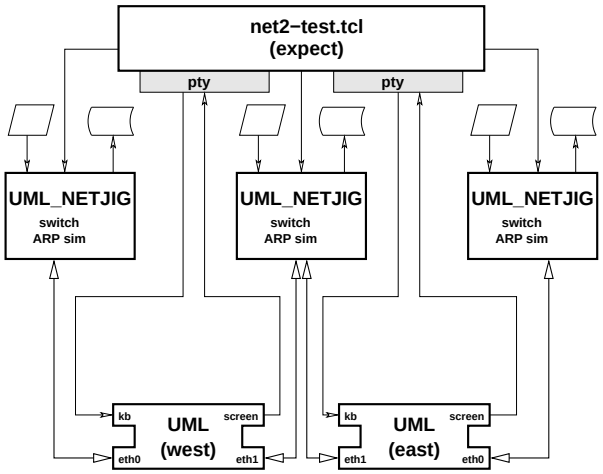

Figure 8: NetJig for multiple machines

References

[Atk95a] R. Atkinson. RFC 1825: Security architecture for the Internet Protocol, August 1995. Obsoleted by RFC2401 [KA98a]. Status: PROPOSED STANDARD.

[Atk95b] R. Atkinson. RFC 1826: IP authentication header, August 1995. Obsoleted by RFC2402 [KA98b]. Status: PROPOSED STANDARD.

[Atk95c] R. Atkinson. RFC 1827: IP encapsulating security payload (ESP), August 1995. Obsoleted by RFC2406 [KA98c]. Status: PROPOSED STANDARD.

[Bec01] Kent Beck. *eXtreme Programming: explained*. Addison-Wesley, 2001.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header

compression, February 1999. Status: PROPOSED STANDARD.

[HC98] D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE), November 1998. Status: PROPOSED STANDARD.

[KA98a] S. Kent and R. Atkinson. RFC 2401: Security architecture for the Internet Protocol, November 1998. Obsoletes RFC1825 [Atk95a]. Status: PROPOSED STANDARD.

[KA98b] S. Kent and R. Atkinson. RFC 2402: IP authentication header, November 1998. Obsoletes RFC1826 [Atk95b]. Status: PROPOSED STANDARD.

[KA98c] S. Kent and R. Atkinson. RFC 2406: IP Encapsulating Security Payload (ESP), November 1998. Obsoletes RFC1827 [Atk95c]. Status: PROPOSED STANDARD.

[MSST98] D. Maughan, M. Schertler, M. Schneider, and J. Turner. RFC 2408: Internet Security Association and Key Management Protocol (ISAKMP), November 1998. Status: PROPOSED STANDARD.

[Per96] C. Perkins. RFC 2003: IP encapsulation within IP, October 1996. Status: PROPOSED STANDARD.

[Pip98] D. Piper. RFC 2407: The Internet IP security domain of interpretation for ISAKMP, November 1998. Status: PROPOSED STANDARD.

[TDG98] R. Thayer, N. Doraswamy, and R. Glenn. RFC 2411: IP security document roadmap, November 1998. Status: INFORMATIONAL.