Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

nfsim: Untested code is buggy code

Rusty Russell
IBM Australia, OzLabs
rusty@rustcorp.com.au

Jeremy Kerr
IBM Australia, OzLabs
jk@ozlabs.org

Abstract

The netfilter simulation environment (`nfsim`) allows netfilter developers to build, run, and test their code without having to touch a real network, or having superuser privileges. On top of this, we have built a regression testsuite for netfilter and iptables.

Nfsim provides an emulated kernel environment in userspace, with a simulated IPv4 stack, as well as enhanced versions of standard kernel primitives such as locking and a proc filesystem. The kernel code is imported into the `nfsim` environment, and run as a userspace application with a scriptable command-line interface which can load and unload modules, add a route, inject packets, run iptables, control time, inspect `/proc`, and so forth.

More importantly we can test every single permutation of external failures automatically—for example, packet drops, kmalloc failures and timer deletion races. This makes it possible to check error paths that very rarely happen in real life.

This paper will discuss some of our experiences with `nfsim` and the progression of the netfilter testsuite as new features became available in the simulator, and the amazing effect on development. We will also show the techniques we used for exhaustive testing, and why these should be a part of every project.

1 Testing Netfilter Code

The netfilter code is complicated. Technically, netfilter is just the packet-interception and mangling framework implemented in each network protocol (IPv4, IPv6, ARP, Decnet and bridging code)[3]. IPv4 is the most complete implementation, with packet filtering, connection tracking and full dynamic Network Address Translation (NAT). Each of these, in turn, is extensible: dozens of modules exist within the tree to filter on different packet features, track different protocols, and perform NAT.

There were several occasions where code changes unintentionally broke extensions, and other times where large changes in the networking layer (such as non-linear `skb`s[1]) caused subtle bugs. Network testing which relies on users is generally poor, because no single user makes use of all the extensions, and intermittent network problems are never reported because users simply hit “Reload” to work around any problem. As an example, the Linux 2.2 masquerade code would fail on one in a thousand FTP connections, due to a control message being split over two packets. This was never reported.

[1] `skb`s are the kernel representation of network packets, and do not need to be in contiguous virtual memory.

2 The Existing Netfilter Testsuite

Netfilter had a testsuite from its early development. This testsuite used the ethertap[2] devices along with a set of helper programs; the tests themselves consisted of a series of shell scripts as shown in Figure 1.

Unfortunately, this kind of testing requires root privileges, a quiescent machine (no `ssh`-ing in to run the testsuite!) and a knowledge of shell slightly beyond cut-and-paste of other tests. The result was that the testsuite bit-rotted, and was no longer maintained after 2000.

2.1 Lack of Testing

The lack of thorough testing had pervasive effects on the netfilter project which only became clear as the lack was remedied. Most obviously, the quality of the code was not all that it could have been—the core functionality was solid, but the fringes contained longstanding and subtle bugs.

The less-noticed effect is the fear this knowledge induces in the developers: rewrites such as TCP window tracking take years to enter the kernel as the developers seek to slowly add users to test functionality. The result is a cycle of stagnation and patch backlog, followed by resignation and a lurch forward in functionality. It's also difficult to assess test coverage: whether users are actually running the changed code at all.

Various hairy parts of the NAT code had not been significantly altered since the initial implementation five years ago, and there are few developers who actually understand it: one of these, Krisztian Kovacs, found a nasty, previously unnoticed bug in 2004. This discovery caused Rusty to revisit this code, which in turn prompted the development of `nfsim`.

[2] ethertap devices are virtual network interfaces that allow userspace programs to inject packets into the network stack.

3 Testsuite Requirements

There are several requirements for a good testsuite here:

- It must be trivial to run, to encourage developers and others to run it regularly;
- It must be easy to write new tests, so non-core developers can contribute to testing efforts;
- It must be written in a language the developers understand, so they can extend it as necessary;
- It must have reasonable and measurable coverage;
- It should encourage use of modern debugging tools such as valgrind; and
- It must make developers *want* to use it.

4 The New Testsuite—nfsim

It was a long time before the authors had the opportunity to write a new testsuite. The aim of `nfsim` was to provide a userspace environment to import netfilter code (from a standard kernel tree) into, which can then be built and run as a standalone application. A command-line interface is given to allow events to be simulated in the kernel environment. For example:

- generate a packet (either from a device or the local network stack); or

```
tools/intercept PRE_ROUTING DROP 2 1 > $TMPFILE &
sleep 1

tools/gen_ip $TAP0NET.2 $TAP1NET.2 100 1 8 0 55 57 > /dev/tap0

if wait %tools/intercept; then :
else
  echo Intercept failed:
  tools/rcv_ip 1 1 < $TMPFILE
  exit 1
fi
```

Figure 1: Shell-script style test for old netfilter testsuite

- advance the system time; or
- inspect the kernel state (e.g., through the `/proc/` file system).

Upon this we can build a simple testsuite.

Figure 2 shows a simple configure-build-execute session of `nfsim`.

Help text is automatically generated from docbook XML comments in the source, which also form the man page and printable documentation. There is a "trivial" XML stripper which allows building if the required XML tools are not installed.

When the simulator is started, it has a default network setup consisting of a loopback interface and two ethernet interfaces on separate networks. This basic setup allows for the majority of testing scenarios, but can be easily reconfigured. Figure 3 shows the default network setup as shown by the `ifconfig` command.

The presence of this default network configuration was a decision of convenience over abstraction. It would be possible to have no interfaces configured at startup, but this would require each test to initialise the required network environment manually before running. From further experience, we have found that the significant majority of tests do not need to alter the default network setup.

Although the simulator can be used interactively, running predefined `nfsim` test scripts allows us to automate the testing process. At present, a netfilter regression testsuite is being developed in the main netfilter subversion repository.

To assist in automated testing, the builtin `expect` command allows us to expect a string to be matched in the output of a specific command that is to be executed. For example, the command:

```
expect gen_ip rcv:eth0
```

will expect the string "`rcv:eth0`" to be present in the output the next time that the `gen_ip` command (used to generate IPv4 packets) is invoked. If the expectation fails, the simulator will exit with a non-zero exit status. Figure 4 shows a simple `nfsim` test which generates a packet destined for an interface on the simulated machine, and fails if the packet is not seen entering and leaving the network stack.

```
$ ./configure --kerneldir=/home/rusty/devel/kernel/linux-2.6.12-rc4/
...
$ make
...
$ ./simulator --no-modules
core_init() completed
nfsim 0.2, Copyright (C) 2004 Jeremy Kerr, Rusty Russell
Nfsim comes with ABSOLUTELY NO WARRANTY; see COPYING.
This is free software, and you are welcome to redistribute
it under certain conditions; see COPYING for details.
initialisation done
> quit
$
```

Figure 2: Building and running `nfsim`

Note that there's a helpful `test-kernel-source` script in the `nfsim-testsuite/` directory. Given the source directory of a Linux kernel, builds `nfsim` for that kernel and runs all the tests. It has a simple caching system to avoid rebuilding `nfsim` unnecessarily.

During early development, a few benefits of `nfsim` appeared.

Firstly, compared to a complete kernel, build time was very short. Aside from the code under test, the only additional compilation involved the (relatively small) simulation environment.

Secondly, 'boot time' is great:

```
$ time ./simulator < /dev/null
real    0m0.006s
user    0m0.003s
sys     0m0.002s
```

4.1 The Simulation Environment

As more (simulated) functionality is required by netfilter modules, we needed to "bring in" more code from the kernel, which in turn depends on further code, leading to a large amount of dependencies. We needed to decide which code was simulated (reimplemented in `nfsim`), and which was imported from the kernel tree.

Reimplementing functionality in the simulator gives us more control over the "kernel." For example, by using simulated notifier lists, we are able to account for each register and deregister on all notifier chains, and detect mismatches. The drawback of reimplementation is that more `nfsim` code needs to be maintained; if the kernel's API changes, we need to update our local copy too. We also need to ensure that any behavioural differences between the real and simulated code do not cause incorrect test results.

Importing code allows us to bring in functionality 'for free,' and ensures that the imported functionality will mirror that of the kernel. However, the imported code will often require support in another area, meaning that further functionality will need to be imported or reimplemented.

For example, we were faced with the decision to either import or reimplement the IPv4 routing code. Importing would guarantee that we would deal with the 'real thing' when it came

```
> ifconfig
lo
        addr: 127.0.0.1  mask: 255.0.0.0  bcast: 127.255.255.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0

eth0
        addr: 192.168.0.1  mask: 255.255.255.0  bcast: 192.168.0.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0

eth1
        addr: 192.168.1.1  mask: 255.255.255.0  bcast: 192.168.1.255
        RX packets: 0 bytes: 0
        TX packets: 0 bytes: 0
```

Figure 3: Default network configuration of `nfsim`

```
# packet to local interface
expect gen_ip rcv:eth0
expect gen_ip send:LOCAL {IPv4 192.168.0.2 192.168.0.1 0 17 3 4}
gen_ip IF=eth0 192.168.0.2 192.168.0.1 0 udp 3 4
```

Figure 4: A simple `nfsim` test

time to test, but required a myriad of other components to be able to import. We decided to reimplement a (very simple) routing system, having the additional benefit of increased control over the routing tables and cache.

Generic functions, or functions strongly tied to kernel code were reimplemented. We have a single `kernelenv/kernelenv.c` file, which defines primitives such as `kmalloc()`, locking functions and lists. The kernel environment contains around 1100 lines of code.

IPv4 code is implemented in a separate module, with the intention of making a 'pluggable protocol' structure, with an IPv6 implementation following. The IPv4 module contains around 1700 lines of code, the majority being in routing and checksum functions.

Ideally, the simulation environment should be as clean and simple as possible; adding complexity here may cause problems when the code under test fails.

4.2 Interaction with Userspace Utilities

The most often-used method of interacting with netfilter code is through the `iptables` command, run from userspace. We needed some way of providing this interface, without either modifying iptables, or reimplementing it in the simulator.

To allow iptables to interface with netfilter code under test, we've developed a shared library, to be `LD_PRELOAD`-ed when running an unmodified `iptables` binary. The shared library intercepts calls to `{set,get}sockopt()`, and diverts these calls to the simulator.

4.3 Exhaustive Error Testing

During netfilter testing with an early version of `nfsim`, it became apparent that almost all of the error-handling code was not being exercised. A trivial example from `ip_tables.c`:

```
counters = vmalloc(countersize);
if (counters == NULL)
        return -ENOMEM;
```

Because we do not usually see out-of-memory problems in the simulator (nor while running in the kernel), the error path (where `counters` is `NULL`) will almost certainly never be tested.

In order to test this error, we need the `vmalloc()` to fail; other possible failures may be due to any number of possible external conditions when calling these 'risky' functions (such as `copy_{to,from}_user()`, semaphores or skb helpers).

Ideally, we would be able to simulate the failure of these risky functions in every possible combination.

One approach we considered is to save the state of the simulation when we reach a point of failure, test one case (perhaps the failure), restore to the previous state, then test the other case (success). This left us with the task of having to implement checkpointing to save the simulator state; while not impossible, it would have been a large amount of work.

The method we implemented is based on `fork()`. When we reach a risky function, we `fork()` the simulator, test the error case in the child process, and the normal case in the parent. This produces a binary tree of processes, with each node representing a risky function. To prevent an explosion in the number of processes, the parent blocks (in `wait()`) while the child executes[3].

The failure decision points are handled by a function named `should_i_fail()`. This handles the process creation, error testing and failure-path replay; the return value indicates whether or not the calling function should fail. Figure 5 shows the `nfsim` implementation of `vmalloc`, an example of a function that is prone to failure. The single (string) argument to `should_i_fail()` is unique per call site, and allows `nfsim` to track and replay failure patterns.

The placement of `should_i_fail()` calls needs to be carefully considered—while each failure test will increase test coverage, it can potentially double the test execution time. To prevent combinatorial increase in simulation time, `nfsim` also has a `should_i_fail_once()` function, which will test the failure case once only. We have used this for functions whose failure does not necessarily indicate an error, for example `try_module_get()`.

When performing this exhaustive error testing, we cannot expect a successful result from the test script; if we are deliberately failing a memory allocation, it is unreasonable to expect that the code will handle this without problems. Therefore, when running these failure tests, we don't require a successful test result, only that the code will handle the failure gracefully (and not cause a segmentation fault, for example). Running the simulator under valgrind[1] can be useful in this situation.

If a certain failure pattern causes unexpected problems, the sequence of failures is printed to allow the developer to trace the pattern, and can be replayed using the `--failpath`

[3] Although it would be possible to implement parallel failure testing on SMP machines by allowing a bounded number of concurrent processes.

```
struct sk_buff *alloc_skb(unsigned int size, int priority)
{
        if (should_i_fail(__func__))
                return NULL;

        return nfsim_skb(size);
}
```

Figure 5: Example of a risky function in `nfsim`

command-line option, running under valgrind or a debugger.

One problem that we encountered was the use of iptables while in exhaustive failure testing mode: we need to be able to `fork()` while interacting with iptables, but can not allow both resulting processes to continue to use the single iptables process. We have solved this by recording all interactions with iptables up until the `fork()`. When it comes time to execute the second case, a new iptables process is invoked, and we replay the recorded session. However, we intend to replace this with a system that causes the iptables process to fork with the simulator.

Additionally, the failure testing is very time-consuming. A full failure test of the 2.6.11 netfilter code takes 44 minutes on a 1.7GHz x86 machine, as opposed to 5 seconds when running without failure testing.

At present, the netfilter testsuite exercises 61% of the netfilter code, and 65% when running with exhaustive error checking. Although the increase in coverage is not large, we are now able to test small parts of code which are very difficult to reliably test in a running kernel. This found a number of long-standing failure-path bugs.

4.4 Benefits of Testing in Userspace

Because `nfsim` allows us to execute kernel code in userspace, we have access to a number of tools that aren't generally available for kernel development. We have been able to expose a few bugs by running `nfsim` under valgrind.

The GNU Coverage tool, `gcov`[2], has allowed us to find untested areas of netfilter code; this has been helpful to find which areas need attention when writing new tests.

Andrew Trigell's `talloc` library[4] gives us clean memory allocation routines, and allows for leak-checking in kernel allocations. The 'contexts' that `talloc` uses allows developers to identify the source of a memory leak.

5 Wider Kernel Testing: `kernsim`?

The `nfsim` technique could be usefully applied to other parts of the kernel to allow a Linux kernel testsuite to be developed, and speed quality kernel development. The Linux kernel is quite modular, and so this approach which worked so well for netfilter could work well for other sections of the kernel.

Currently `nfsim` is divided into `kernelenv`, `ipv4` and the `netfilter` (IPv4) code. The

first two are nfsim-specific testing implementations of the kernel equivalents, such as `kmalloc` and `spin_lock`. The latter is transplanted directly from the kernel source.

The design of a more complete `kernsim` would begin with dividing the kernel into other subsystems. Some divisions are obvious, such as the SCSI layer and VFS layer. Others are less obvious: the slab allocator, the IPv4 routing code, and the IPv4 socket layer are all potential subsystems. Subsystems can require other subsystems, for example the IPv4 socket layer requires the slab allocator and the IPv4 routing code.

For most of these subsystems, a simulated version of the subsystem needs to be written, which is a simplified canonical implementation, and contains additional sanity checks. A good example in `nfsim` is the packet generator which always generates maximally nonlinear `skb`s. A configuration language similar to the Linux kernel 'Kconfig' configuration system would then be used to select whether each subsystem should be the simulator version or imported from the kernel source. This allows testing of both the independent pieces and the combinations of pieces. The latter is required because the simulator implementations will necessarily be simplified.

The current `nfsim` commands are very network-oriented: they will require significant expansion, and probably introduction of a namespace of some kind to prevent overload.

5.1 Benefits of a `kernsim`

It is obvious to the `nfsim` authors that wider automated testing would help speed and smooth the continual redevelopment which occurs in the Linux kernel. It is not clear that the Linux developers' antipathy to testing can be overcome, however, so the burden of maintaining a `kernsim` would fall on a small external group of developers, rather than being included in the kernel source in a series of `test/` subdirectories.

There are other possibilities, including the suggestion by Andrew Tridgell that a host kernel helper could allow development of simple device drivers within `kernsim`. The potential for near-exhaustive testing of device drivers, including failure paths, against real devices is significant; including a simulator subsystem inside `kernsim` would make it even more attractive, allowing everyone to test the code.

6 Lessons Learnt from `nfsim`

Nfsim has proven to be a valuable tool for easy testing of the complex netfilter system. By providing an easy-to-run testsuite, we have been able to speed up development of new components, and increase developer confidence when making changes to existing functionality. Netfilter developers can now be more certain of any bugfixes, and avoid inadvertent regressions in related areas.

Unfortunately, persuading some developers to use a new tool has been more difficult than expected; we sometimes see testsuite failures with new versions of the Linux kernel. However, we are confident that `nfsim` will be adopted by a wider community to improve the quality of netfilter code. Ideally we will see almost all of the netfilter code covered by a nfsim test some time in the near future.

Adopting the simulation approach to testing is something that we hope other Linux kernel developers will take interest in, and use in their own projects.

Downloading `nfsim`

`nfsim` is available from:

`http://ozlabs.org/~jk/projects/nfsim/`

Legal Statement

References

[1] Valgrind Developers. Valgrind website. `http://valgrind.org/`.

[2] Free Software Foundation. GCOV — a Test Coverage Program. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[3] Netfilter Core Team. Netfilter/iptables website. `http://netfilter.org`.

[4] Andrew Tridgell. talloc website. `http://talloc.samba.org/`.