Paranoid Penguin

Running Network Services under User-Mode Linux, Part III

Mick Bauer

Abstract

Fine-tune and firewall your UML guest systems.

In the last two Paranoid Penguin columns, I walked you through the process of building a virtual network server using User-Mode Linux. We built both host and guest kernels, obtained a prebuilt root filesystem image, configured networking on the host, and when we left off last month, we finally had booted our guest kernel with bridged networking, ready for configuration, patching and server software installation.

This month, I tie up some loose ends in our example guest system's startup and configuration, show you the uml_moo command, demonstrate how to write firewall rules on your UML host system, offer some miscellaneous security tips and give some pointers on creating your own root filesystem image. And, can you believe we will have scratched only the surface of User-Mode Linux, even after three articles? Hopefully, we'll have scratched deeply enough for you to be off to a good start!

Guest System Configuration

You may recall that last time we set up bridged networking on our host, creating a local tunnel interface called uml-conn0 that we bridged to the host system's “real” eth0 interface. If you don't have last month's column, my procedure was based on the one by David Cannings (see the on-line Resources). When we then started up our host (User-Mode) kernel, we mapped a virtual eth0 on the guest to uml-conn0 via a kernel parameter, like so:

```
umluser@host$ ./debkern ubd0=debcow,debroot
"root=/dev/ubda eth0=tuntap,uml-conn0
```

The last parameter, obviously, contains the networking magic: eth0=tuntap,uml-conn0. It can be translated to “the guest kernel's eth0 interface is the host system's tunnel/tap interface uml-conn0”. This is important to understand; to the host (real) system, the guest's Ethernet interface is called uml-conn0, but to the guest system itself, its Ethernet interface is plain-old eth0.

Therefore, if you run an iptables (firewall) rule set on either host or guest (I strongly recommend you do so at least on the host), any rules that use interface names as sources or targets must take this difference in nomenclature into account. We'll discuss some example host firewall rules shortly, but we're not quite done with guest-kernel startup parameters yet.

Going back to that startup line, we've got definitions of our virtual hard drive (ubd0, synonymous with ubda), our path to virtual root and, of course, our virtual Ethernet interface. But what about memory?
On my OpenSUSE 10.1 host system, running a UML Debian guest with the above startup line resulted in a default memory size of about 29MB -- pretty puny by modern standards, especially if I want that guest system to run real-world, Internet-facing network services. Furthermore, I've got an entire gigabyte of physical RAM on my host system to allocate; I easily can spare 256MB of RAM for my guest system.

To do so, all I have to do is pass the parameter mem=256M to the guest kernel, like so:

```
umluser@host$ ./debkern mem=256M ubd0=debcow,debroot
  +root=/dev/ubda eth0=tuntap,uml-conn0
```

Obviously enough, you can specify however much more or less than that as you like, and you can allocate different amounts of RAM for multiple guests running on a single host (perhaps 128M for your virtual DNS server, but 512M for your virtual Web server, for example). Just be sure to leave enough non-guest-allocated RAM for your host system to do what it needs to do.

Speaking of which, you'll save a lot of RAM on your host system by not running the X Window System, which I've always recommended against running on hardened servers anyhow. The X server on my test host uses around 100MB, with actual desktop managers requiring more. On top of this, the X Window System has a history of security vulnerabilities with varying degrees of exploitability by remote attackers (remember, a “local” vulnerability ceases being local the moment any non-local user starts a shell).

**Managing COW Files**

If, as I recommended last month, you run your UML guest with a Copy on Write (COW) file, you may be wondering whether your UML guest-kernel startup line is the only place you can manage COW files. (A COW file is created automatically when you specify a filename for one in your ubd0=... parameter.)

Actually, the uml-utilities package includes two standalone commands for managing COW files: uml_moo and uml_mkcow. Of the two, uml_moo is the most likely to be useful to you. *You can use uml_moo to merge all the filesystem changes contained in a COW file into its parent root filesystem image.*

For example, if I run the example UML guest kernel startup command described earlier, and from within that UML guest session I configure networking, apply all the latest security patches, install BIND v9 and configure it and finally achieve a “production-ready” state, I may decide that it's time to take a snapshot of the UML guest by merging all those changes (written, so far, only into the file debcow) into the actual filesystem image (debroot). To do so, I'd use this command:

```
umluser@host$ uml_moo ./debcow newdebroot
```

The first argument you specify to uml_moo is the COW file you want to merge. Because a COW file contains the name of the filesystem image to which it corresponds, you don't have to specify this. Normally, however, you should specify the name of the new filesystem image you want to create.

My example uml_moo command, therefore, will leave the old root filesystem image debroot intact (maybe it's also being used by other UML guests, or maybe I simply want to preserve a clean image), creating a new filesystem named newdebroot that contains my fully configured and updated root filesystem.
If I want to do a hard merge, however, which replaces the old filesystem image with the merged one (with the same filename as before), perhaps because my hard disk is too full for extra image files, I'd instead use `uml_moo -d ./debcow` (the -d stands for destructive merge).

### iptables and UML

Whether you chroot your User-Mode guests, and whether you use SELinux, depends on how deep you want your layers of security to go and how much time and effort you're able to expend. However, I strongly recommend that on any Internet-facing, bridged User-Mode Linux system, you use `iptables` on your UML host to restrict your guest systems' network behavior.

On the one hand, if your UML system already resides outside a firewall in a DMZ network (as should any Internet server), you're already protecting your internal network from the possibility of a network server compromise. However, there's really no good reason not to take the opportunity also to use UML-host `iptables` rules to reduce the ability of an attacker to use one compromised UML guest to attack other UML guests, the UML host itself or other systems in your DMZ network.

There are two categories of rules I strongly recommend you consider. First, anti-IP-spoofing rules can help ensure that every packet sent by each guest bears the source IP address you actually assigned to that guest, and not a forged (spoofed) source IP. These are low-maintenance rules that you'll have to think about only at setup time, unless for some reason you change a guest system's IP address.

Suppose you have a UML system whose IP address is 10.1.1.10 and whose tun/tap interface is (from the host's perspective) `uml-conn0`. The anti-spoofing rules you install on the UML host might therefore look like that shown in Listing 1.

#### Listing 1. Anti-IP-Spoofing Rules

```
iptables -A FORWARD -m physdev --physdev-in uml-conn0
   -s ! 10.1.1.10 -j LOG --log-prefix "Spoof from uml-conn0"
iptables -A FORWARD -m physdev --physdev-in uml-conn0
   -s ! 10.1.1.10 -j DROP
```

The first rule logs the spoofed packets; the second one actually drops them. As you may know, the LOG target doesn't cause packets to cease being evaluated against subsequent `iptables` rules, but the DROP target does, so the LOG rule must come before the DROP rule.

Due to space constraints, I can't launch into a primer on how to write `iptables` rules or how they're managed on your Linux distribution of choice. But, I can talk about the bridge-specific magic in Listing 1: the physdev `iptables` module and the `--physdev-in` parameter.

Usually, we use `iptables`' -i and -o flags to denote which network interface packets are received and sent from, respectively. However, when writing `iptables` rules on a system doing bridged networking, we need to be a bit more precise, especially when we're also using tun/tap interfaces, as eth0 then takes on a different role than in normal Layer 3 (routed) networking.

Therefore, where we might normally use `-i uml-conn0` in a rule, on a bridging host, we should instead use `-m physdev --physdev-in uml-conn0`. Similarly, instead of `-o uml-conn0`, we'd use `-m physdev --physdev-out uml-conn0`. As with other module invocations, you need only one instance of `-m`
physdev if a given iptables rule uses both the --physdev-in and --physdev-out rules.

After setting up a pair of anti-IP-spoofing rules, you also should create a set of “service-specific” rules that actually govern how your guest system may interact with the rest of the world, including other guest systems and the host itself.

Remember that in our example scenario the guest system is a DNS server. Therefore, I'm going to enforce this logical firewall policy:

1. The UML guest may accept DNS queries (both TCP and UDP).
2. The UML guest may recurse DNS queries against upstream (external) servers.
3. The UML guest may send its log messages to a log server (called logserver).
4. The UML host may initiate SSH sessions on the UML guest.

Listing 2 shows iptables commands that could enforce this policy.

**Listing 2. Service Rules for the UML Guest**

```
iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A FORWARD -m physdev --physdev-out uml-conn0 -p udp --dport 53 -m state --state NEW -j ACCEPT
iptables -A FORWARD -m physdev --physdev-out uml-conn0 -p tcp --dport 53 -m state --state NEW -j ACCEPT
iptables -A FORWARD -m physdev --physdev-in uml-conn0 -p udp --dport 53 -d ! 10.1.1.0/24 -m state --state NEW -j ACCEPT
iptables -A FORWARD -m physdev --physdev-in uml-conn0 -p tcp --dport 53 -d ! 10.1.1.0/24 -m state --state NEW -j ACCEPT
iptables -A FORWARD -m physdev --physdev-in uml-conn0 -p udp --dport 514 -d logserver -m state --state NEW -j ACCEPT
iptables -A FORWARD -j LOG --log-prefix "Forward Dropped by default"
iptables -A FORWARD -j DROP
iptables -A OUTPUT -d 10.1.1.10 -p tcp --dport 22 -m state --state NEW -j ACCEPT
```

Listing 2 has two parts: a complete set of FORWARD rules and a single OUTPUT rule. Because, logically speaking, UML guest systems are “external” to the UML host's kernel, interactions between UML guests and each other, and also interactions between UML guests and the rest of the world, are handled via FORWARD rules. Interactions between UML guests and the underlying host system, however, are handled by INPUT and OUTPUT rules (just like any other interactions between external systems and the host system).

Because all of my logical rules except #4 are enforced by iptables FORWARD rules, Listing 2 shows my UML host's complete FORWARD table, including an initial rule allowing packets associated with
already-approved sessions, and a final pair of “default log & drop” rules. Note my use of the physdev module; I like to use interface-specific rather than IP-specific rules wherever possible, as that tends to make it harder for attackers to play games with IP headers.

The last rule in Listing 2 should, in actual practice, appear somewhere in the middle of a similar block of OUTPUT rules (beginning with an allow-established rule and ending with a default log/drop rule pair), but I wanted to illustrate that where the source or destination of a rule involves the UML host system, you can write an ordinary OUTPUT or INPUT rule (respectively) rather than a FORWARD rule.

Because your UML host is acting as an Ethernet bridge, you can write still-more-granular and low-level firewall rules -- even filtering by MAC addresses, the ARP protocol and so forth. But for that level of filtering, you'll need to install the ebttables command. iptables rules of the type I've just described should, however, suffice for most bastion-host situations.

### Miscellaneous Security Notes

If you patched your UML host's kernel with the SKAS patch, you've already got reasonably good assurance that an attacker who compromises a UML guest won't be able to do much, if anything, on the host system. However, I'm not one to argue against paranoia, so I also recommend you chroot your UML guest system. This is described in detail on the UML Wiki (see Resources). And, what about shell access to your UML guests? There are various ways to access “local consoles”. You get one automatically when you start your UML guest from a UML host shell manually -- after your UML kernel loads, you'll be presented with a login prompt.

That doesn't do you much good if you start your UML guest automatically from a script, however. The “Device Inputs” page on the User-Mode Linux home page (see Resources) describes how to map UML guest virtual serial lines to UML host consoles. For me, however, it's easiest simply to install SSH on my UML guest system, configure and start its SSH daemon, and create a firewall rule that allows connections to it only from my UML host.

Generally speaking, you want to use the same security controls and tools on your UML guest (tripwire, chrooted applications, SELinux, tcpwrappers and so on) as you would on any other bastion server.

### Building Your Own Root Filesystem Image

Describing in detail the process of building your own root filesystem image from scratch would require its own article (one which I may yet write). Suffice it to say, the process is all but identical to that of creating your own bootable Linux CD or DVD, without the final step of burning your image file to some portable medium. There are three major steps:

1. Create an empty filesystem image file with `dd`.
2. Format the image file.
3. Mount it to a directory via loopback.
4. Install Linux into it.
The first three steps are the easiest. To create a 1GB ext3 image file, I'd run the commands shown in Listing 3 as root.

**Listing 3. Making and Mounting an Empty Filesystem Image**

```
dd if=/dev/zero of=./mydebroot bs=1024K count=1000
mkfs.ext3 ./mydebroot
mkdir /mnt/debian
mount -o loop ./mydebroot /mnt/debian
```

Installing Linux into this directory gets a bit more involved, but if you've got a SUSE host system, the Software module in YaST includes a wizard called “Installation into Directory”. Like other YaST modules, this is an easy-to-use GUI.

Similarly, if you run Debian, you can use the command `debootstrap`. See Michael McCabe and Demetrios Dimatos' handy article “Installing User Mode Linux” for detailed instructions on using debootstrap to populate your root filesystem image.

See the UML Wiki for some pointers to similar utilities in other distributions. The Linux Bootdisk HOWTO (see Resources), although not specific to UML, is also useful.

**Conclusion**

I hope you're well on your way to building your own virtual network servers using User-Mode Linux! The two most important sources of UML information are the UML home page and the UML Wiki (see Resources). Those and the other Web sites mentioned in this piece should help you go much further with User-Mode Linux than I can take you in an introductory series of articles like this. Have fun, and be safe!

**Resources for this article:** [http://www.linuxjournal.com/article/9457](http://www.linuxjournal.com/article/9457).