Porting SSLSERV: A Descent Into Madness

Adam Thornton
Sine Nomine Associates
August 11, 2004

Introduction

Actually, it wasn't all that bad. This document intends to be a fairly straightforward description of the process of getting a Debian/390 system to act as an SSLSERV host. I presume familiarity with Linux and with VM TCP/IP.

Setting Up the Build Environment

The first step was David Boyes telling me I should put SSLSERV on Debian. "Easy for you to say," is the sort of thing one thinks at one's boss, but doesn't actually say aloud.

The motivations here are pretty simple: annual maintenance for SLES is a very steep sum if all you really want is SSL-wrapped TCP/IP connections. Sine Nomine sells a VM-tailored Debian distribution now. We've been very involved in porting Debian-Installer to S/390—having it be an officially supported architecture in Debian-Installer means that S/390 will be a fully supported platform when Debian "Sarge" is released as the new "stable" Debian distribution, currently targeted for September 15. We certainly intend to produce a "Sarge"-based, VM-tailored distribution once it's officially released.

Thus, the choice of distribution was easy. The initial implementation (through version 0.3) was also suboptimal: what is actually installed underneath the SSLSERV system is Debian "Sid," also known as "unstable," rather than "Sarge," "testing." That's the choice I made because not all of the packages that Debian-Installer requires were in "testing" when I started. With the release of Debian-Installer RC1, it was possible to build "Sarge" using Debian-Installer, which is what I did. "Sarge" underlies 0.4 and later releases. Debian "Woody"—current "stable"—would also work, actually, but it's going to be obsolete in a couple months, and I wanted a fairly future-proof substrate. ("Unstable" in the Debian world is generally about as reliable as other vendors' "*N*.1" releases; to get *really* dodgy, bleeding-edge packages, you need to go to "experimental." "Testing" is almost always fairly good, and since "Sarge" is aiming for a September 15 release as "Stable" it's quite solid.)

The next stage was to go to the *TCP/IP Planning and Customization* guide. Chapter 23 (at least, of the z/VM 4.4 guide, which is what I was working from since 4.4 is what we're running) is about the SSL server and is absolutely indispensable for getting this thing to work. According to the doc, the code I need is on 4TCPIP40 493. I tried to link it, and was told there was no such disk. "Aha," thought I, "SFS installation." Sure

enough, VMSYS:4TCPIP40.BINARY holds the appropriate RPMBIN files. Reading the README seems to indicate that the VMSLSDB set is intended for SLES 7, and VMSLDSB is for SLES 8. The difference seems to be whether or not the IUCV driver needs to be replaced with a newer one, since VMSLDSB doesn't require that step. The default "Sarge" kernel is Linux 2.4.26 with quite recent drivers, so I assumed that I wouldn't need any patches to IUCV. Both the source and binary RPMs are about 1.6MB each, so the total amount of code is small. VMSLDSB INSTALL contains setup instructions, which seem straightforward.

The next step was to set up what would be the SSLSERV machine. I knew that the SSLSERV code was quite small: under 10 MB, even unpacked. So I just needed to install a minimal Debian system. I knew that I could do this in 250 cylinders with a little room to spare. Working from VMSLDSB INSTALL, I devided to make this my 151-disk, and to use a real 152-disk (not VDISK) as swap; I chose 100 cylinders for that. Judging from the parmline in the install document, I was supposed to access 152 as /dev/dasda, 151 as /dev/dasdb, and 203—which is a 1-cylinder CMS minidisk, containing a file that appears to Linux as a DIAG-accessed ext2 partition—as /dev/dasdc. Debian-Installer will not currently format non-ECKD disks, but can access them just fine. So I did a basic Debian-Installer installation, detecting, in order, 151, 152, and 203, and within minutes had a running system.

Next I created a new build host; this step wasn't really necessary, but I wanted a machine that hadn't been upgraded piecemeal with chunks of various Debian distribution levels. It's got a 1500-cylinder 150-disk, which is plenty for a development environment.

The next step was to grab the IBM SSL code and take a look at what I had. I used good old FTP from my build host to the VM stack to grab the RPMBIN files. Now, as you probably know, Debian doesn't use rpms. No big deal: I used apt-get to install "alien", and *alien –t* ***rpmfile*** turned each RPM into a tarball. I unpacked each of those and snooped around. Basically, the stuff you build locally is inside vmssld-1.24.19.tgz. It's easy enough to unpack it, and it's just a couple of C files.

Building and Linking the Linux vmssl Daemon

The first attempt at a build failed miserably. After scratching my head for a few minutes, I realized that the build process creates a kernel module, and therefore I'd need properly initialized kernel sources. So, off to get the kernel source tree, and then run "make menuconfig; make dep" in order to set up the appropriate symlinks to the right architectures. That did the trick, and I was able to build the vmsock.o module. It even insmodded OK; now, granted, I had to force it because the kernel level reported and the kernel tree used weren't quite in sync (2.4.26-2 versus 2.4.26-1-s390), but since I was going to be running all kinds of OCO code anyway, I didn't care much.

The actual VMSSL implementation depends on having the module loaded, but also on some OCO libraries shipped with it. These belong in /opt/vmssl/lib, and the binary goes

in /opt/vmssl/bin. LD_LIBRARY_PATH is set to look in /opt/vmssl/lib first. Well, that was pretty easy to do. Then I tried to start vmssl and immediately the app crashed and burned. Judicious use of "ldd" showed me that I was looking for libstdc++-libc6.1-2.so.3 and libstdc++-libc6.2-2.so.3.

Debian's never heard of these. Apt-cache shows me some older libstdc++ libraries, but nothing *that* ancient. At this point, I shrieked in despair and cursed the VM SSL developers. After a few minutes of howling and calling David, I was reminded by him that this seemed similar to what we found when trying to install the TSM client on SLES8, and the answer was to install the compatibility libraries.

That's good advice, but there's no such thing for Debian. However, I then trucked over to rpmfind.net and looked for "libstdc++-libc6.1-2.so.3" and found it in Fedora Core Development. Again, alien was my friend; armed with the RPM, I had it unpacked in short order.

The Compatibility Libraries

I'm now going to take a non-chronological digression—I didn't actually package the compatibility stuff up as a .deb until much later, but it makes sense to talk about it here. Using *alien –d* creates a .deb package. That wouldn't install, because one pair of libraries conflicted with libstdc++-5 on the system. That's OK, though: alien generated the debian rules to build the package too, so I just removed those libraries and ran dpkg-build to create a .deb with just the libraries I needed.

Now, because the compatibility libraries are GPLed, we have to provide source for the version we used if anyone demands it. If they do, they're going to get the source RPM. I looked at it with an eye towards doing a proper Debian package. No thank you. It's an entire ancient GCC (2.95.3) rebuild, with 433 patches (yes, *really*: that's what *ls *.patch | wc* told me) applied. I presume that rpmbuild, on a suitably-equipped system, would yield the RPM I converted with alien. I haven't ever built these compatibility libraries from scratch on Debian, and frankly I don't intend to.

It turns out that the reason these dependencies exist is vmssl's reliance on a very old GSKit: vmssl is no spring chicken, and it was linked against GSKit back in the SLES 7 days. Both vmssl and GSKit are badly, in my opinion, in need of recompilation with a more up-to-date compiler and standard libraries.

End of digression.

Preparing the SSLSERV Virtual Machine

Now "ldd" reported that all links were resolved. Running "vmssl" seemed to make something happen: I got 104 spawned "vmssl" processes: 100 worker threads (which is

documented as the default), plus a master, and a couple others for something else. Time to move it over to the actual SSLSERV system. I IPLed SSLSERV and immediately got a disabled wait state. Turns out that it expects to IPL from 201, in stark contrast to the 151 recommended by the installation guide. No big deal: you can set IPLDEVICE in SYSTEM DTCPARMS (or *nodeid* DTCPARMS, or SSLSERV DTCPARMS) in z/VM 4.4. You can also set TCPUSERID, which is handy (albeit undocumented), since I wanted to test with the TCPIP2 stack, not TCPIP. Apparently these knobs do not exist under z/VM 4.3, according to Thomas Kern, who has been an extremely helpful tester throughout this process. In that case you need a local TCPIP DATA on SSLSERV's 191-disk, copied there (and modified on the fly with the correct TCPUSERID and IPLDEVICE) by an exit exec. (Presumably, for true correctness, you'd do this anyway if you needed to couple to a secondary stack; myself, I use the undocumented TCPUSERID parameter.) Anyway, it didn't take long to IPL SSLSERV from its 151, and a minimal but functional Debian system soon appeared.

On the SSLSERV system, I copied the /opt/vmssl tree over from my build system, copied the compatibility .deb, ran *dpkg –i* to install it, modified /etc/fstab to mount the 203-disk at /opt/vmssl/parms, and was ready to test it. Since the executable started with a suitably modified LD_PRELOAD, I hadn't lost any functionality. Now came the scary part: destroy the existing network functionality. From the INSTALL document, it seemed that I just had to turn off /etc/init.d/networking and any services that depended on it. Debian uses a tool called rc-update.d; basically, I renamed each of inetd, networking, and ssh to *service*.DISABLED and ran *rc-update.d* ***service*** *remove*. I also needed to get rid of automated loading of network interfaces, which was a simple matter of moving /etc/network/interfaces out of the way, and remove any network parameters from /etc/modutils. This done, I ran /sbin/update-modules.

Then it was time to include automated startup of the vmssl daemon. I copied the startup script (supplied in /opt/vmssl/bin) into /etc/init.d, and used *rc-update.d* to tell it to run at priorities S95 and K05 in runlevel 2 (Debian uses runlevel 2, unlike SuSE's 5, as its default). Now I was ready to shut down; any changes from this point forward would have to be done from the console (ick) or by mounting the disk on another Linux image and modifying it there.

So that's what I did.

Testing the Server

The init script failed, spectacularly, to run. Debian doesn't have *startproc*; instead it uses *start-stop-daemon*. However, I was able to log in at the console and manually start VMSSL. From TCPMAINT, with a suitable TCPIP DATA (to set TCPIP2) I was able to run SSLADMIN Q and get a reasonable report. I used the VM SSL tools to create a testing certificate for myself.

Next I played with PROFILE TCPIP for TCPIP2. After some struggling (although this part is well-documented in *TCP/IP Planning and Customization*) I had SSLADMIN listening to port 992 in secure mode, using the testing certificate I had created. Using x3270 with SSL enabled allowed me to get a VM login screen with a little lock in the upper right, so all was well.

Preparing and Improving the Releases

Now I had to go back and fix the startup script. It's not pretty. I eventually left most of the old code alone (just fixing a couple of "=" vs. "==" errors—I'm guessing it was written by a Rexx coder) but added a distribution check to see whether I was running on SuSE, Red Hat, or Debian (I intend to extend this for Tao and Slackware in the near future). In the "Debian" branch, I initially ran start-stop-daemon to background the process itself. That worked—and so I could IPL SSLSERV and have it come up properly, which was a huge step—but I lost logging. Nevertheless, at this point, it worked, and I released it to my testers as version 0.1.

Version 0.2 moved the backgrounding into a vmssl-wrapper program which also allowed setting log files and running in either the foreground or the background. I also realized that there was no reason to preserve the 151 device address, so I changed /etc/zipl.conf to use 152, 201, and 203, and made the disk into my 201. This allowed me to remove the IPLDEVICE parameter from the DTCPARMS file, which meant that if the user was using the primary TCPIP stack, no changes to DTCPARMS would be necessary. Although log files were saved on the Linux host, SSLADMIN LOG did not retrieve them.

Version 0.3 was a minor update: I had used Perl to build the wrapper, but decided that C would be more portable, since I didn't know if other distributions could be counted on to have a Perl interpreter in a default installation. So I rewrote vmssl-wrapper in C, and, at Thomas Kern's request, documented using SWAPGEN to put swap on VDISK rather than on real DASD.

Version 0.4 replaced "Sid" with "Sarge," installed via Debian-Installer RC1. This certainly makes us the first official "Sarge" S/390 distribution, for whatever that's worth. This revision was slightly annoying, in that I also had to rebuild my development machine as "Sarge" to get the right levels of the toolchain so that the vmsock module would load without a panic. This also works around the minor irritation that the SSLADMIN LOG function relies on a hardcoded path to a log file, and SSLADMIN LOG now works as intended.

Future Directions

What's next? Well, maybe repackaging VMSSL as a binary .deb. This isn't actually as much of a no-brainer as it appears: after all, once it's applied, your Linux host is never

useful as a Linux machine again, so having everything under dpkg control is of questionable utility. I'd also like to fix the startup script so that it runs on Red Hat, Tao, and Slackware as well as Debian and SuSE; at that point, rewriting the INSTALL document to be vendor-neutral would be nice.

A nicely-packaged single-3480-tape install should be easy to create, and ought to satisfy customers who would like physical media. We could also put it on CD quite easily; both of these would be an improvement over a hundred-megabyte download.

More excitingly, let's consider some of the limitations of the current SSL implementation. The biggest, obviously, is that there's no hardware crypto API support. Writing an actual driver would be hard—but maybe we don't have to. How does SSLSERV do its magic? Evidently, via an IUCV connection. We have the IUCV driver, and the IUCV interface is well-documented. OpenSSL 9.7 can interface with the ibmca driver to use hardware crypto functions. So what we'd need to do, in order to replace all the OCO code that IBM has put into SSLSERV, and to enable hardware support, would be to replace the SSL implementation that vmssl uses with OpenSSL compiled with engine support, and with vmssl turning on the engine (if present) before creating its child threads. Of course, that means we'd have to implement the vmssl daemon from scratch—but how hard would that be? We know it talks over IUCV, and it functions essentially identically to stunnel: it accepts an SSL-wrapped connection, unwraps it, and hands it back.

We'd need, of course, a decent AF_IUCV implementation. The folks at BMC have a first cut at that up on their FTP site, and if nothing else, it's a place to start. We'd probably need to look at the raw data flow to determine how commands and data are differentiated to the Linux vmssl process, which could be tricky. The answer to "how hard would that be?" would be—and this is only a guess—somewhere between "pretty" and "darn" hard. Of course, a proper IUCV implementation would have implications far wider than the SSLSERV daemon.

It would unquestionably be a lot of work. The VM team says that they would like to rebuild the server with modern tools and incorporate zSeries hardware crypto support. If those two enhancements were made, the impetus to spend a lot of time reinventing this particular wheel would be greatly diminished.