

Tom's Networking

How To: Hacking the Linksys NSLU2 – Part 2

By Jim Buzbee – August 10 2004

Introduction

In my previous article, I showed how to get a command prompt on the NSLU2 by using a hidden option and editing the password file in an external system. In this article, we'll continue exploring the box with the goal of adding new functions to make it even more useful.

Since the box runs a version of Linux, there are a whole host of software packages that one could install on the system. One could envision expanding the web server on the box for custom applications, or perhaps installing a ftp server or a secure shell server to replace the telnet server. My first goal was to install a different network file system.

As delivered by Linksys, the only support provided is for SMB—the network file system protocol that is native to Microsoft Windows. Since my home network is mostly composed of UNIX-based machines such as Apple's Mac OS X and Linux, a natural extension for me is NFS, the network file system native to most UNIX systems.

But before we can start installing new packages, there are a few difficulties to overcome. The first thing we'll have to do is set up a development environment, since we'll be building the required packages. The rest of this article assumes that you have followed the previous article and have enabled telnet on your NSLU2.



Disclaimer: It goes without saying, but I'll say it anyway. Exploring the NSLU2 by looking at its internal file structures using any method that doesn't modify its code should leave your warranty intact. But modifying the NSLU2 in any way will void your warranty.

TomsNetworking, Tom's Guides Publishing and I are not responsible for any damage that the information in this article may cause to your NSLU2 or any data it manages.

So download a copy of the current firmware before you start, and don't go trying to get help from Linksys if you break it.

Determining the Target Processor

Linux is the most natural environment for developing software for another Linux system so I'll do this from my Mandrake development box. Unlike most Linux development systems, the NSLU2 is not x86 based, so we'll have to install a cross-compilation toolchain.

The first step in setting up a cross-compilation toolchain is determining the target processor. Reviews I had read indicated that the processor was an Intel IXP420 Network Processor. But to tell you the truth, that didn't mean much to me, since it sounded more like a marketing term.

Since we can now log into the box, we can poke around and see what Linux thinks the processor is. The `/proc` filesystem is invaluable for this. One file of interest is `/proc/cpuinfo`. Reading it is easy :

```
# cat /proc/cpuinfo
Processor       : XScale-IXP425/IXC1100 rev 1 (v5b)
BogoMIPS       : 131.48
Features        : swp half thumb fastmult edsp

Hardware       : Intel IXDP425 Development Platform
Revision       : 0000
Serial         : 0000000000000000
```

That didn't tell us much more than we already knew, but it does validate previous reports regarding the IXP425. To get further, we'll have to take a different tack. A handy utility on most UNIX systems is `file`, which examines a file and tells you what kind of file it is based on the initial data in the file.

The NSLU2 doesn't include the `file` program, but it does include `ftp`. So using `ftp`, we can move an executable over to our development platform. Try moving over our favorite little program, `telnet.cgi` from the directory `/home/httpd/html/Management`. Once this is on your development system run `file` on it to see what it thinks:

```
# file telnet.cgi
telnet.cgi: ELF 32-bit MSB executable, ARM, version 1 (ARM),
for GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
```

That's better. Now we know we're dealing with an ARM CPU so we'll have to install an ARM cross compilation toolchain.

A Toolchain Shortcut

The thought of installing a cross-compilation toolchain instills fear in the hearts of many developers. The idea of compiling a compiler and C library and linker, plus support libraries and utilities is just too much. Fortunately in the age of the Internet, there has to be someone out there who has already gone through the pain and is sharing their effort.

For gcc cross-compilers that someone is Dan Kegel. Dan has a terrific [script](#) that starts from scratch. It downloads the required sources, patches them as necessary and builds a complete toolchain—and can do it for many different processors. I've used Dan's scripts to build a cross-compilation toolchain for the MIPS based Linksys WRT54G and it was a breeze. I've heard from at least one person that Dan's scripts will build a toolchain that produces NSLU2 compatible binaries. I highly recommend Dan's tools, but for the purposes of this article I'm going to take a short cut.

Linksys appears to be struggling a bit with the release of source code as required under the GPL license. Many of their devices run Linux internally, but their GPL releases differ greatly. Some include a complete build environment while others like the NSLU2 have a less than complete release.

The CD that accompanies the NSLU2 has a great deal of source, but no toolchain. Fortunately, Linksys has another Linux-based device—their [WRV54G](#) Wireless-G VPN Broadband Router—that appears to be based on the same Intel IXP425 development board. Since they have released a toolchain for that product, we'll use it. [Download](#) the 125 MB (!) file and untar it in your home directory.

For me, the top of the tree looks like :

```
/home/jbuzbee/gpl_code_2.03
```

Inside this directory you'll find a file called toolchain.tgz. It looks like this was designed to be untarred from the root directory but I choose to untar it in place so as to keep all of my build environment in my home directory. After untarring it, you'll find the required compilers, linkers, include files, etc.

The binaries reside in the `usr/local/openrg/armlsaeb/bin` directory and all have a prefix of `armv4b-hardhat-linux-`. I put that directory in my `$PATH` so I don't have to type out the full directory tree each time I compile.

Testing the Compiler

Let's try out the compiler to make sure it generates a good binary. The first thing any self-respecting C programmer writes is "Hello World", so I created a file called `hello.c`:

```
#include
```

```
main( int argc char *argv[] )  
{
```

```
    printf("Hello world!\n");  
}
```

then compiled it :

```
# export PATH=home/jbuzbee/gpl_code_2.03/usr/local/openrg/armsaeb/bin:$PATH  
# armv4b-hardhat-linux-gcc hello.c -o hello
```

and checked the resulting file type :

```
# file hello  
hello: ELF 32-bit MSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,  
dynamically linked (uses shared libs), not stripped
```

Looks good. Now move the file over to your NSLU2 by ftp'ing it starting from your NSLU2 box. Remember to do a binary transfer and to mark the file as executable. Once it's there, try it out :

```
# ./hello  
Hello World!
```

It works! Now we have the ability to create and run arbitrary programs on our NSLU2. Make sure that anything you want to keep gets put on your NSLU2 hard drive. By default, the home directory for the root account is on a ram disk. Create a new directory in the normal Linux way and then put your new binary in it. For me it's :

```
# mkdir /share/hdd/data/jim/nslu2  
# mv hello /share/hdd/data/jim/nslu2/
```

Modifying and Building the Portmapper

Now we can move on to bigger and better things. It may seem a big step to go from "Hello World" to NFS, but it's not too bad. First we need to install and setup the Linux source tree in order to access the required include files. The source tree can be found in the snapgear tar file included on the CD distributed with the NSLU2.

First untar the snapgear tar file in your build area. Since we are not going to be building the kernel, we'll take a shortcut setting up for an ARM architecture. Create and execute the following script from within the snapgear/linux-2.4.x directory

```
#!/bin/bash  
  
rm -f include/asm  
rm -f include/asm-arm/proc  
rm -f include/asm-arm/arch
```

```
( cd include; ln -s asm-arm asm )  
( cd include/asm-arm; ln -s proc-armv proc )  
( cd include/asm-arm; ln -s arch-ixp425 arch )
```

Now we should be set up for more complex builds. NFS is based on a Remote Procedure Call architecture. In order for RPC to function, it needs a helper daemon called the portmapper. I found portmapper source code [here](#), and source for an NFS server [here](#). After untarring the source trees you should end up with portmap_4 directory, a snapgear directory and a nfs-server-2.2beta47 directory along with the original gpl_code_2.03 directory.

Next let's build the portmapper. I had to make a few changes to the Makefile to simplify the build. First, I commented out the line : HOSTS_ACCESS= -DHOSTS_ACCESS, to reduce an external library dependence. Everywhere I saw a reference to libwrap.a I also commented it out for the same reason. For a more secure server, these can be put back, but additional libraries will be needed.

I then modified the CFLAGS line to point to my snapgear linux source tree :

```
CFLAGS = $(COPT) -O $(NSARCHS) $(SETPGRP) -I /home/jbuzbee/snapgear/linux-  
2.4.x/include/
```

Finally, in the top of the makefile, I added the following line in order to invoke the cross compiler:

```
CC=/home/jbuzbee/gpl_code_2.03/usr/local/openrg/armsaeb/bin/armv4b-hardhat-linux-  
gcc
```

In addition to the makefile changes, I needed to make a small source change. In the file portmap.c, there's a perror function defined that overrides the standard perror. I changed the argument definition from const char to __const char so as to match the stdio.h definition in our cross compilation tree.

Now, all that is required is a "make". This produces a number of warnings, but the build should run to completion. Next strip the binary to save memory for our little box:

```
# armv4b-hardhat-linux-strip portmap
```

Adding the NFS Daemons

One compile down, one more to go. Now we'll tackle the NFS daemons themselves. The build process for the NFS daemons requires the execution of a script called BUILD. I could find no option in this script for cross-compilation, so I just let it do its thing and fixed the results after. A number of questions are asked during the run of the script. Answer everything with the default except for the question:

Do you want to protect mountd with HOST ACCESS?

Answer "no" to this question since we don't have the required library. After the configuration runs to completion, you'll need to edit the resulting Makefile to fix it up for our cross compiler. Near the top of the makefile, there are three variables for CC, AR and RANLIB. Add the prefix for your cross-compiler. For example, my CC now looks like this:

```
CC = /home/jbuzbee/gpl_code_2.03/usr/local/openrg/armsaeb/bin/armv4b-hardhat-linux-gcc
```

Next, we need to add an include path to the CFLAGS line. Mine looks like so:

```
CFLAGS = -g -O -D_GNU_SOURCE -I /home/jbuzbee/snapgear/linux-2.4.x/include/
```

The final change we need to make is in the file nfsd.h. Near the top, add an additional include that is needed for our environment:

```
#include <time.h>
```

Once this is done, a simple make will build the two required binaries: rpc.mountd and rpc.nfsd. Stripping these binaries doesn't seem to reduce their size. Now we're ready to try NFS out.

Move the three binaries; portmap, rpc.mountd, rpc.nfsd over to the NSLU2. I decided to put my new system level binaries under the conf partition in order to keep them separate from my data. I created three new directories under /share/hdd/conf—bin, etc, and rc.d, mirroring the directories in the normal tree.

The binaries go to the bin directory, startup scripts go to the rc.d directory, and configuration files go to the etc directory. In our case, the only configuration file needed is the exports file. NFS requires an exports file to tell the NFS daemon which directories to export and to whom among other things. I created an exports file that looked like this:

```
/share/ 192.168.1.100 (rw, insecure)
/share/ 192.168.1.103 (rw )
/share/ 192.168.1.102 (rw )
```

The insecure option is for my Mac OS X box which uses insecure network ports. You'll obviously need to create a exports file that matches your network needs, so for more options, consult the standard NFS [documentation](#).

Checking it Out

Now we need to set everything up cleanly. The NFS server expects to find the exports file in the top level /etc directory, so create a symbolic link to our new exports file in the /share/hdd/conf/etc directory :

```
# ln -s /share/hdd/conf/etc/exports /etc/exports
```

Next, we need nice startup scripts for all of our new daemons. On the NSLU2, copy the rc.samba script from the /etc/rc.d directory to the new rc.d directory and modify it twice: once to create an rc.portmap script and once to create an rc.nfsd script. The rc.nfsd script should start the rpc.mountd and the rpc.nfsd.

Finally, build one script to do it all. Create the symbolic link, execute the portmap script and execute the nfsd script in that order. This script will need to be run each time the box is booted. As of yet, I have no way to automatically run startup scripts when the box boots.

Now, run the do-it-all script and try it out. A ps -ax from the NSLU2 should show three new processes running—portmap, rpc.mountd, and rpc.nfsd. Error and status messages will show up in the file /var/log/messages.

From a client machine try to mount the NSLU2. On my Mandrake system I executed the following as root:

```
# mkdir /mnt/nslu2
# mount 192.168.1.70:/share/ /mnt/nslu2/
```

On my Mac OS X box, I use the "Connect to Server" option of the Finder with a server format of:

```
nfs://192.168.1.70/share
```

If you have any issues starting the daemons, or in mounting, check the /var/log/messages file on the NSLU2. Note that having NFS on the box does not affect SMB, they can coexist peacefully.

Now that we've shown we can build a fairly complex service for the box, the sky's the limit. A FTP server might be a good next step. Or maybe an embedded SSH server such as [dropbear](#) would be useful. Let me know how it goes!

NOTE: As I wrote this article, I started from scratch and re-executed all of my original steps in order to make sure I hadn't forgotten anything. As usual, my development box was short on disk space, so naturally I NFS mounted the disk from the NSLU2. I did all of my untarring, compilations, editing, etc. on my NFS mounted drive. It performed flawlessly!

Tip: If you want to skip all this stuff and just want a binary that you can load onto your NSLU2 (you'll still need to edit the etc/exports file), you can download the ~57KB file from <http://www.nyx.net/~jbuzbee/nslu2-nfs-0.1.tar.gz>.

In Part 3, I'll show you how the NSLU2 can keep you entertained as well as serve your files. In the meantime, you can follow my adventures on my [Linux on the NSLU2 page](#).