Cross Compiling for Prehistoric Systems

By Mark Bereza

Figure 1. First users of Linux Kernel 2.6.30, pictured above.

Rationale

When you find an embedded device perfectly preserved in metaphorical cyber amber, do you donate it to the nearest museum or try to hack it? If the device is practically prehistoric—perhaps created by the Plateosauri whose stubby arms couldn't reach the F5 and F10 keys and never bothered to ship the system with GDB—what do you do?

Discovering Features About Your Relic

Figure 2. Fossil records suggest this tool was a favorite of Neanderthal hackers. An elegant weapon, for a less civilized age.

In order to correctly configure our toolchain for the device, you will need to do some research/reconnaissance to determine its various features, including:

- processor name/model
- presence of an FPU
- instruction set architecture
- ABI (if applicable)
- Linux kernel version
- libc type and version
- GCC version

This guide will use an ARMv5 embedded system running Linux kernel version 2.6.30 as an example target. While a comprehensive guide for discovery of each of these features for whatever embedded device you found in King Tut's tomb is outside the scope of this page, the following general strategies might prove useful:

1. Discovery of the device name, model, and manufacturer should come first. You should be particularly interested in the processor on the device. Often this can be found by simply inspecting the device for any labels or branding. If the device has an FCCID, search for this ID on the FCC ID Database. Try Googling any values found via barcodes or QR codes.
2. If you're able to find out the name of the device or its processor, look up the datasheet for the device. These can often be found on the manufacturer's website. Datasheets can help us find out what hardware is running on the device and can often tell us the instruction set architecture and whether or not the device supports hardware floating point.
3. The other information requires access to the device's filesystem or at least a binary you know runs on the device. If you have access to the filesystem, perform string searches for key words using grep:

```
grep -rn '/root/of/unpacked/filesystem' -e 'keyword'
```

    I. If `uname -a` doesn't work, you might be able to find information about the Linux distribution/version under `/etc/` in a file containing the word "version," "release," or "issue."

    II. Processor information can often be found under `/proc/`. Check to see if the `/proc/` directory contains a file named 'version"; if it exists, if often contains useful information.

    III. The libc type and version can be found by seeing what `/lib/`

`libc.so.X` symlinks to. As an example, the `/lib/` directory on the aforementioned ARMv5 device contains `libuClibc-0.9.30.so`, which indicates that it's using uClibc version 0.9.30. Other common C libraries include glibc, eglibc, and musl.

IV. Performing a string search for "gcc" on /lib/libgcc_s.so.X will often help you find not only the GCC version used to build the binaries and filesystem, but will also give you information about the toolchain that was used. Going back to our example device, we found the following string in `/lib/libgcc_s.so.1`:
`/here/workdir/factory/build_armv5l-timesys-linux-uclibcgnueabi/gcc-4.3.3/gcc-4.3.3/libgcc/../gcc/config/arm/lib1funcs.asm`
This tells us that the device is running ARMv5l, that the Timesys factory was used to generate the toolchain, the GCC version is 4.3.3, the ABI is EABI, and the libc is uClibc.

4. If you have access to a binary that you know runs on the device (ex: downloaded firmware), you can discover information about the device using readelf.

   I. To find the name of the architecture, run:
   `readelf -h /path/to/binary | grep -i "Machine:"`

   II. To find the endianess, run:
   `readelf -h /path/to/binary | grep -i "Data:"`

   III. To find the ABI, run:
   `readelf -h /path/to/binary | grep -i "Flags:"`

   or:

   `readelf -A /path/to/binary | grep -i "Attribute Section:"`

   IV. To find the architecture version, run:
   `readelf -A /path/to/binary | grep -i "Tag_CPU_arch"`

Setting Up Prehistoric Linux in a Virtual Machine

To hack the dinosaurs, you must first think like a dinosaur. Think to yourself, "what OS would an ancient megalizard use?" Ubuntu 12.04 is the answer. As an added benefit, Ubuntu 12.04 is also the latest version of Ubuntu that supports Buildroot 2009.08, the latest version of Buildroot that supports older kernels (> 2.6.X). If your system isn't quite so ancient, you might be able to get away with a newer version of Buildroot, but for this guide, we'll move forward with this version. First, install it onto a virtual machine. Begin by obtaining an image of Ubuntu 12.04 here (the image linked was reconstructed from fossil molds but should be sufficiently accurate for our purposes).

Once your VM is setup, be sure to run:

```
sudo apt-get update
sudo apt-get upgrade
```

It is *crucial* that we get the most cutting edge updates for this prehistoric Ubuntu. Oh, and it's also useful so that you obtain the certificates that `wget` will look for when downloading packages for our toolchain.

Installing the Buildroot of my Father and His Father Before Him

You can download Buildroot 2009.08 from here. In the spirit of historical accuracy, the following instructions will use a technology appropriate for that era, also known as CLI. The original meaning of the acronym has been lost to time, but scholars speculate it might've stood for "Caveman Language Interface." Whatever the case may be, this nigh-hieroglyphic text seems to get the job done:

```
wget http://buildroot.org/downloads/buildroot-2009.08.tar.gz
tar xzf buildroot-2009.08.tar.gz
rm -f buildroot-2009.08.tar.gz
cd buildroot-2009.08
```

Generating a Cross Compile Toolchain

Configuring Buildroot

To actually generate a cross compile toolchain for ARM, run the following:

```
sudo apt-get install automake bison flex gettext g++ libncurses-
dev texinfo
make menuconfig
```

Note: `automake, bison, flex, gettext, g++, libncurses-dev`, and `texinfo` are all Buildroot dependencies.

If all goes well, you will now encounter what at first glance might appear to be a graphic interface. DO NOT BE FOOLED. It is simply a hack meant to make text with various background colors look like a GUI. From this menu, you can customize the toolchain Buildroot will generate in various ways. Which options you should select will depend on what you discovered during the recon phase. For our example system, the following options were selected (the rest left default):

Target Architecture	arm
Target Architecture Variant	arm926t
Target ABI	EABI
Target Options → Atmel Device Support	Yes
Target Options → Atmel Device Support → Board Support for the Atmel AT91 Range of Microprocessors	Yes
Target Options → Atmel Device Support → Allow All ARM Targets	No
Target Options → Atmel Device Support → AT91 Device	Atmel AT91SAM9263 Microprocessor
Toolchain → Kernel Header Options	Linux 2.6.29.X Kernel Headers
Toolchain → uClibc C Library Version	uClibc 0.9.30
Toolchain → Thread Library Debugging	Yes
Toolchain → Build GDB Debugger for the Target	Yes
Toolchain → Enable Toolchain Locale/i18n Support?	Yes
Toolchain → Use Software Floating Point by Default	Yes
Package Selection for the Target → BusyBox	No
Package Selection for the Target → strace	Yes
Package Selection for the Target → Networking → tcpdump	Yes

An explanation of the options selected:

- From the recon conducted, we knew that the device was using an Atmel AT91SAM9263, which is an arm926t chip.
- Although we knew that the device is running Linux 2.6.30, we intentionally selected the slightly older (didn't even think that was possible) 2.6.29.X version to be safe since kernel headers are backwards compatible and we're unsure about the minor version number Buildroot will generate.
- We want to make sure we enable thread library debugging, that way we are able to debug multithreaded processes with the GDB we generate.
- Locale support ended up being a "gotcha" in this case since we found that the libc generated without that option enabled would not generate certain symbols needed by existing binaries on the device.
- "Use Software Floating Point by Default" was selected since we know the device does not have an FPU. If you're not sure about the floating point capabilities of your device, this is a safe bet.
- We don't want to build BusyBox because we don't need it, but we do want GDB, strace, and tcpdump for debugging purposes.

From here exit `menuconfig` and save your unholy configuration.

Downloading Dependencies

For antique versions of Buildroot, some of the repositories Buildroot will try to download packages from will be dead. Thus, before you can run `make`, you may need to manually download a few archives that Buildroot needs but cannot download automatically since the mirrors it's looking for are now being used as fuel for cars. In our case, we needed to download the following packages:

- gdb-6.8.tar.bz2
- zlib-1.2.3.tar.bz2
- strace-4.5.18.tar.bz2
- fakeroot_1.9.5.tar.gz
- genext2fs-1.4.tar.gz

Inevitably the mirrors above will also disintegrate into dust, so depending on how far into the future you're reading this from, you might need to find the files yourself. If the archive containing the files is named "fossils," that's usually a good sign:

Once downloaded, move all these files into Buildroot's download directory using:

```
mv -t /path/to/buildroot-2009.08/dl/ gdb-6.8.tar.bz2 \
                                     zlib-1.2.3.tar.bz2 \
                                     strace-4.5.18.tar.bz2 \
                                     fakeroot_1.9.5.tar.gz \
                                     genext2fs-1.4.tar.gz
```

Building the Toolchain

Now, run:

```
make
```

You might want to pull out your sundial at this point since this step will take anywhere between 3 and 7 moons to complete.

If you've been following along with our example, you'll eventually encounter the following build error:

```
mkdir /home/buildroot/buildroot-2009.08/build_arm/makedevs-host
cp target/makedevs/makedevs.c /home/buildroot/buildroot-2009.08/build_arm/makedevs-host
/usr/bin/gcc -Wall -Werror -O2 /home/buildroot/buildroot-2009.08/build_arm/makedevs-
host/makedevs.c -o /home/buildroot/buildroot-2009.08/build_arm/makedevs-host/makedevs
/home/buildroot/buildroot-2009.08/build_arm/makedevs-host/makedevs.c: In function 'main':
/home/buildroot/buildroot-2009.08/build_arm/makedevs-host/makedevs.c:366:6: error: variable
'ret' set but not used [-Werror=unused-but-set-variable]
cc1: all warnings being treated as errors
make: *** [/home/buildroot/buildroot-2009.08/build_arm/makedevs-host/makedevs] Error 1
```

It appears that our ancestors, in their ancient wisdom, decided that enabling `-Werror` was a great idea when compiling a C file with unresolved warnings. But you're not afraid of a little historical revisionism, are you? From the Buildroot root directory, open this file using:

```
vi build_arm/makedevs-host/makedevs.c
```

Note: it is critical that you use `vi` to edit the file to get the full Neanderthal experience.

Find the return 0; statement at the end of `main()` (located on line 534) and change it to `return ret;`. Save your changes and run:

```
make
```

This time, the build should stop with the following output:

```
makedevs: line 41: regular file '/home/buildroot/buildroot-
2009.08/project_build_arm/uclibc/root/bin/busybox' does not exist: No such file or directory
-rw-rw-r-- 1 buildroot buildroot 6979584 Oct 19 15:55 /home/buildroot/buildroot-
2009.08/binaries/uclibc/rootfs.arm.ext2
rm -f /home/buildroot/buildroot-2009.08/project_build_arm/uclibc/.fakeroot*
```

In the tongue of the dinosaurs, a file not found error followed by a `rm -f` command translates to "the build was successful." Just take our word for it. Binaries for your target can be found in `/path/to/buildroot-2009.08/project_build_arm/uclibc/root/usr/bin/.`

Generating (Stone) Tools Using Make

If you want to build additional packages for your target after running make the first time, you might be able to build it using `make <name of binary>`. For example, running `make strace` from the buildroot-2009.08 directory will build strace.

Using the Cross Compile Toolchain to Build Binaries from Source

If Buildroot cannot build the package you want automatically or if you want to build the package statically, you will have to resort to the lost art of manually configuring and building from source, the same way ancient peoples would craft their own clothes out of the hide of the wild beasts they slew.

Unfortunately, there is no universal strategy for accomplishing this as different programs use different build methods. That being said, many utilize the `configure → make → make install` workflow and there are some common steps that will get you most of the way there. We will illustrate this with an example. In particular, we will go through the process of statically building GDB for our ARM system from source.

```
wget http://ftp.lanet.lv/ftp/GNU/gdb/gdb-6.8.tar.bz2
tar xjf gdb-6.8.tar.bz2
rm -f gdb-6.8.tar.bz2
cd gdb-6.8
mkdir build-gdb
cd build-gdb
export PATH=$PATH:/path/to/buildroot-2009.08/build_arm/staging_dir/usr/bin/
export CFLAGS=-static
export CPPFLAGS=-static
export LDFLAGS=-static
../configure --enable-static --disable-shared --host=arm-linux-uclibcgnueabi
make
```

Naturally, you should replace all the "`/path/to`"s with the actual location of these directories on your system.

These steps are common to the cross compiling process for many programs. These include prepending the location of your cross compile toolchain to your PATH, enabling static linking via `--enable-static` and `--disable-shared` (configure step) and XFLAGS=`-static` (make step), and setting the host to the prefix of your generated toolchain (`--host=...`).

Keep in mind that GDB and other tools have their own dependencies and if your toolchain was not built with these dependencies this sort of manual build will fail. For example, GDB requires the `ncurses` library, which Buildroot does not build for the target by default. If you're just trying to build a static version of a package Buildroot supports, you can simply select the package in Buildroot's `menuconfig` and rerun `make` to build all of its dependencies into the toolchain. If the tool you want isn't on Buildroot's package list, you can try finding the dependencies manually and search for them inside `menuconfig` using the "/" key.

On the other hand, if you just want to compile a simple C program you wrote (named foo.c, for example), you would run the following:

```
export PATH=$PATH:/path/to/buildroot-2009.08/build_arm/staging_
dir/usr/bin/
arm-linux-gcc -Wall -g -o foo foo.c
```

Here, GCC flags were set in order to:

1. Enable debug symbols for GDB using `-g`. This is usually a good choice since we'll likely be testing and debugging whatever code we've crafted for our target.
2. Enable all warnings using `-Wall` just to be safe.

How Can I Tell if this Caveman Voodoo Even Works?

While the most robust way to validate the binaries generated via cross compiling is to simply run them on the target platform, you can also perform a quick sanity check using the file command, like so:

```
file /path/to/binary
```

As an example, running `file` on the `gdb` binary created using the method outlined in the previous section produces the following output:

```
gdb: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
statically linked, not stripped
```

This is a good sign since we were indeed targeting little endian ARM and we wanted a statically linked binary.

To be even more thorough, you can run:

```
readelf -hdA /path/to/binary
```

which will print off the entire ELF header, the location of any required shared libraries, and in-depth architecture information. Running this on the same GDB binary produced the following output:

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF32
Data:                              2’s complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           ARM
Version:                           0x1
Entry point address:               0x80d0
Start of program headers:          52 (bytes into file)
Start of section headers:          11315256 (bytes into file)
Flags:                             0x4000002, has entry point, Version4 EABI
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         4
Size of section headers:           40 (bytes)
Number of section headers:         29
Section header string table index: 26
There is no dynamic section in this file.
Attribute Section:                 waeabi
File Attributes
Tag_CPU_name:                      “5T”
Tag_CPU_arch:                      v5TE
Tag_ARM_ISA_use:                   Yes
Tag_THUMB_ISA_use:                 Thumb-1
Tag_ABI_PCS_wchar_t:               4
Tag_ABI_FP_denormal:               Needed
Tag_ABI_FP_exceptions:             Needed
Tag_ABI_FP_number_model:           IEEE 754
Tag_ABI_align_needed:              8-byte
Tag_ABI_align_preserved:           8-byte, except leaf SP
Tag_ABI_enum_size:                 int
```

This output confirms that the binary was built for ARMv5TE with the EABI ABI and has no dynamic section (because it is statically linked).

Conclusion

It's difficult to anticipate what issues you'll run into and the exact process may vary substantially from device to device. As with many things in the field of reverse engineering, persistence is key. Don't be discouraged if you encounter several build errors you don't recognize when running `make`, this is "normal." The Buildroot mailing list is your friend and it's likely someone has already encountered your issue and documented the solution there.

Additionally, make sure to allocate enough time to perform recon on the device ahead of time. The better you understand all the various software/hardware specs of your target, the easier it will be to build a toolchain that will produce programs that actually run on the device. This is especially important since you often won't discover incompatibilities until you actually run the program on the target and changing a single toolchain setting often requires a full rebuild, making iterative approaches extremely time-consuming.

Finally, although persistence is crucial, be careful not to go down unneeded rabbit-holes. Focusing on the big picture - the problem you're actually trying to solve - can aid in this. Do you really need to build a toolchain from scratch? Perhaps someone has already created one for your device. Do you really need this *exact* version of GDB/strace/etc.? If building a specific tool proves too challenging, try a different tool or a different version. Do your binaries really need to be built statically? Perhaps your device already has the needed libraries. If not, you may be able to copy the ones in your toolchain environment onto the device to make a dynamically linked executable work.

Best of luck, and happy hacking!

Trellix

Visit Trellix.com to learn more.

About Trellix

Trellix is a global company redefining the future of cybersecurity. The company's open and native extended detection and response (XDR) platform helps organizations confronted by today's most advanced threats gain confidence in the protection and resilience of their operations. Trellix's security experts, along with an extensive partner ecosystem, accelerate technology innovation through machine learning and automation to empower over 40,000 business and government customers.

Copyright © 2022 Musarubra US LLC

042022-01