IoT: Client Devices

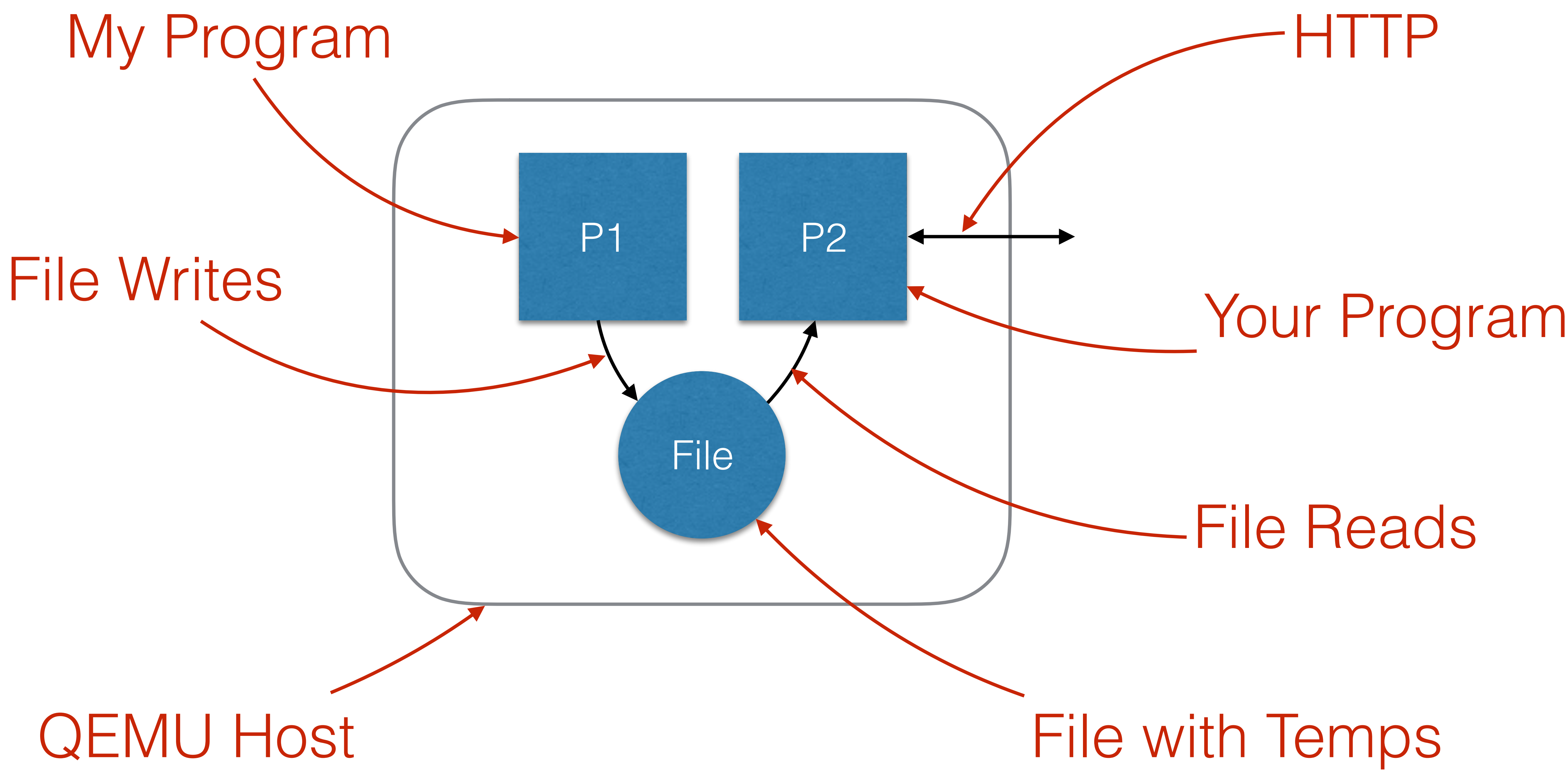
Project (II)

Goal

Emulating a thermostat

- I'll provide a small program to you that writes temperatures to a file (in degrees Celsius)
- You'll pretend that file is an actual thermocouple
- Your thermostat is programmable (no less than 3 different points over a day)
  - Weekends? calendars? weekly programs? extra credit!
- Program remotely via HTTP interface
- Report temperatures and status via HTTP interface
- You'll turn a heater on/off based on the program and reported temperature (you'll write this to a known file with a timestamp)

How will this work?



Requirements

YOUR SYSTEM SHALL

- Read the current temperature from a known file
  - /var/log/temperature
  - Read a single temperature value written to file
  - Float in degrees C
- Turn heat on/off based on program and current temperature
  - /var/log/heater
  - Turn heat on/off by writing to /var/log/heater
  - A single line <action> : <timestamp>
  - action:= <on|off> timstamp:=<posix time of action>

Requirements

YOUR SYSTEM SHALL

- Start a daemon service that can also run from command line
- Process a configuration file
  - Default option, also supplied to program via -c & —config_file flags (e.g. -c <config_file> or —config_file <config_file>)
- Provide a help option (-h or —help)
  - This will print typical help for the application

Requirements

YOUR SYSTEM SHALL

- The configuration file shall configure
  - Service endpoint (e.g. http://<some_host>:8000)
  - Log files (e.g. /my/logfile/here, for program output)
  - Any other config files
- Accept programs via an HTTP interface
  - program up to three different temperatures for a day set at arbitrary times

Requirements

YOUR SYSTEM SHALL

- Report status to an outside process via HTTP
- Report actions to an outside process via HTTP

YOUR SYSTEM MAY:

- Support more extensive programming

IoT: Client Devices

Personal Development Process

What is a Personal Process?

SO YOU NEED A REPEATABLE WAY TO CODE

- What do you set up first?
- How do you design makefiles?
- How do you generally structure your code?
- Do you prototype then insert into a production project?
- Do you use unit tests?
- How do you test the system?
- How do you deliver the developed product?

Why is it Important?

SPEED

- Develop faster; don’t repeatedly solve the same problems
- Understanding your process allows you to develop faster

QUALITY

- Established process leads to higher quality code

REPEATABILITY

- Allows you to repeat and improve

Are you talking about PSP?

Personal Software Process

- This is not that
- PSP is a more dictatorial process
- If you want to adopt this, that’s okay

Your process should be yours

- You should understand it clearly
- It should be second nature

An Example

- When you start a project, start from a template that includes certain files
- Each function is defined in a single file
- Each file has an associated unit test
- Applications consist of discrete libraries, the main function is as simple as possible
- Automated unit tests run against the repository every night
- Automated builds run tests and deploy software to test environment

…and so on.

IoT: Client Devices

Attack Surfaces

What is an attack surface?

THE ATTACKABLE SURFACE OF A SYSTEM

- Anything an attacker can access
- Includes things like configuration files, function arguments, network traffic, music files
- Really, anything the system touches

IoT clients have large attack surfaces

Why is it Important?

How systems can be attacked

- An attack surface describes how attackers will attempt to compromise a system

How systems can be hardened

- Understand the vulnerabilities? you can harden them

What can be neglected

- Just as important!

How to Document?

Not in code, but a document

- The exercise is worth more than documentation
- But you should document so you can review

Pictures are a good thing!

- Make it a simple and clear as possible

What kind of document?

- Doesn’t matter; PDF, MS Word, Wiki, Text, all okay

Example

THE LS COMMAND ON LINUX

- Inputs:
  - various command line options
  - some support user-defined input (—block-size, —color, etc.)
  - what about environment variables? yep! (LS_COLORS)
  - How about the filesystem?

This is the attack surface

Hardening

WE HAVE THE SURFACE DEFINED, NOW HARDEN

- Support different command line options *and combinations*
- Check for well-formed environment variables
- Check buffer lengths
- Check for well-formatted submitted data
- Attackers will submit odd characters, binary code, huge arguments, inconsistent arguments, anything that might break your system

Never ever trust user input!

IoT: Client Devices

Project (I)

Remainder of Course

Large project

- You'll develop parts over the course of the module

One piece at a time

- This way you can test and integrate

Development, Technology

- Rest of the course will have practical talks on developing your client and theoretical talks on system structure

Project: IoT Client

IoT client on ARM

- Not actual hardware
- QEMU Client

Written in C or C++ (this is up to you)

- Examples will be in C
- I will discuss program design in C too
- I won’t go into C++, nor how to install C++ runtimes

Project: IoT Client

Bi-directional communication

- We'll use HTTPS
- You can run on local system

Simple command, reporting protocols

- You'll design this too
- Run over HTTPS

Project: IoT Client

EVALUATE EACH OTHER

- I'll supply rubrics for evaluation
- You'll evaluate your peers

FOUR CATEGORIES

- Design, development, function, security

REMEMBER YOU'LL BE EVALUATED ON THE SYSTEM

- You will deliver the filesystem, kernel, and run script
- Bad passwords? unprotected accounts? don't do it!
- The system is the OS, filesystem, libraries, and your code

Project: IoT Client

Design

- You’ll be evaluated on overall design
- Design of code, not design on paper
- Ease of use and evaluation are important too

Development

- How has the client been developed is important
- Did you use tests? did you use modular programming? is the application inappropriately monolithic?
- Is the code commented and clear? No obfuscated C please!

Project: Iot Client

SECURITY

- You system should be secure
- Strong passwords, good programming practice, understood attack surface
- Kernel should be recent, libraries shouldn’t have known, unprotected flaws

FUNCTION

- It should work!

IoT: Client Devices

Looking over libcurl.so

Let’s use Our RE Skillz

REVERSE ENGINEERING LIBCURL.SO

- Grab it from your guest
- $ scp -P 2222 localhost:/usr/lib/libcurl.so .

Let’s take a look at it.

TAKE A LOOK AT THE HIGH LEVEL STUFF FIRST

- arm-linux-gnueabi-readelf -a libcurl.so
- arm-linux-gnueabi-objdump -d libcurl.so > libcurl.dump
- arm-linux-gnueabi-strings -n 5 libcurl.so > strings.out
- cat strings.out | grep curl > curl-strings.out

This gives you insight into the library.

Does it look legit?

We can see the machine instruction set

- ARM, little-endian

We can see the libraries this links with

- Do we have them all on the system?

We can see exported (and imported) functions

- Check out the **curl_** prefix

Other Metadata

SSL versions? encryption algorithms? websites?

What about the disassembly?

IoT: Client Devices

A Libcurl Example - Setting Up

A Sample Project

First things first

- Remember your personal process?

Create a directory

- Create a .gitignore file
- Create a makefile
- Create an (empty) file named requestor.c

A Sample Project

Create a repository

- $ git init

Create a public repository on GitHub

- Then add the remote reference to your project
- Instructions are on the webpage shown after you create your project

Add the files, commit, and push

- $ git add .
- $ git commit -m’initial import’
- $ git push -u origin master

Make

Make

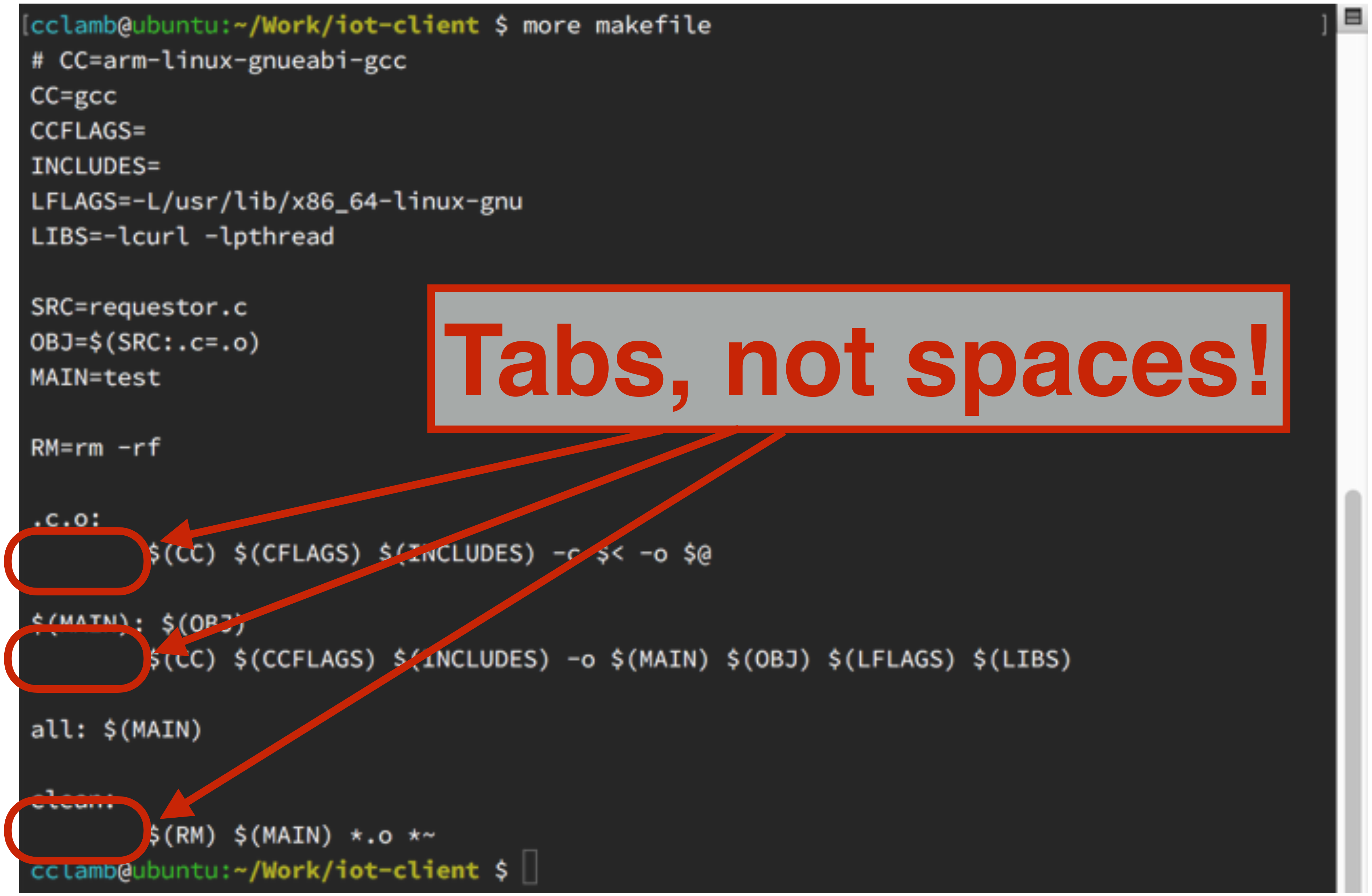
- Older than old, but still useful!

Whitespace sensitive syntax

- Tabs and spaces are treated differently

Many examples on the internet

- Including ones I'll show you



```
cclamb@ubuntu:~/Work/iot-client $ more makefile
# CC=arm-linux-gnueabi-gcc
CC=gcc
CCFLAGS=
INCLUDES=
LFLAGS=-L/usr/lib/x86_64-linux-gnu
LIBS=-lcurl -lpthread

SRC=requestor.c
OBJ=$(SRC:.c=.o)
MAIN=test

RM=rm -rf

.c.o:
	$(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

$(MAIN): $(OBJ)
	$(CC) $(CCFLAGS) $(INCLUDES) -o $(MAIN) $(OBJ) $(LFLAGS) $(LIBS)

all: $(MAIN)

clean:
	$(RM) $(MAIN) *.o *~
cclamb@ubuntu:~/Work/iot-client $
```


Makefile

Kind of cryptic, but very useful! We'll see more later.

IoT: Client Devices

A Libcurl Example - Code

Code!!!

Workstation first

- Test our code
- Easier environment
- Faster test/code cycles
- More stable networking

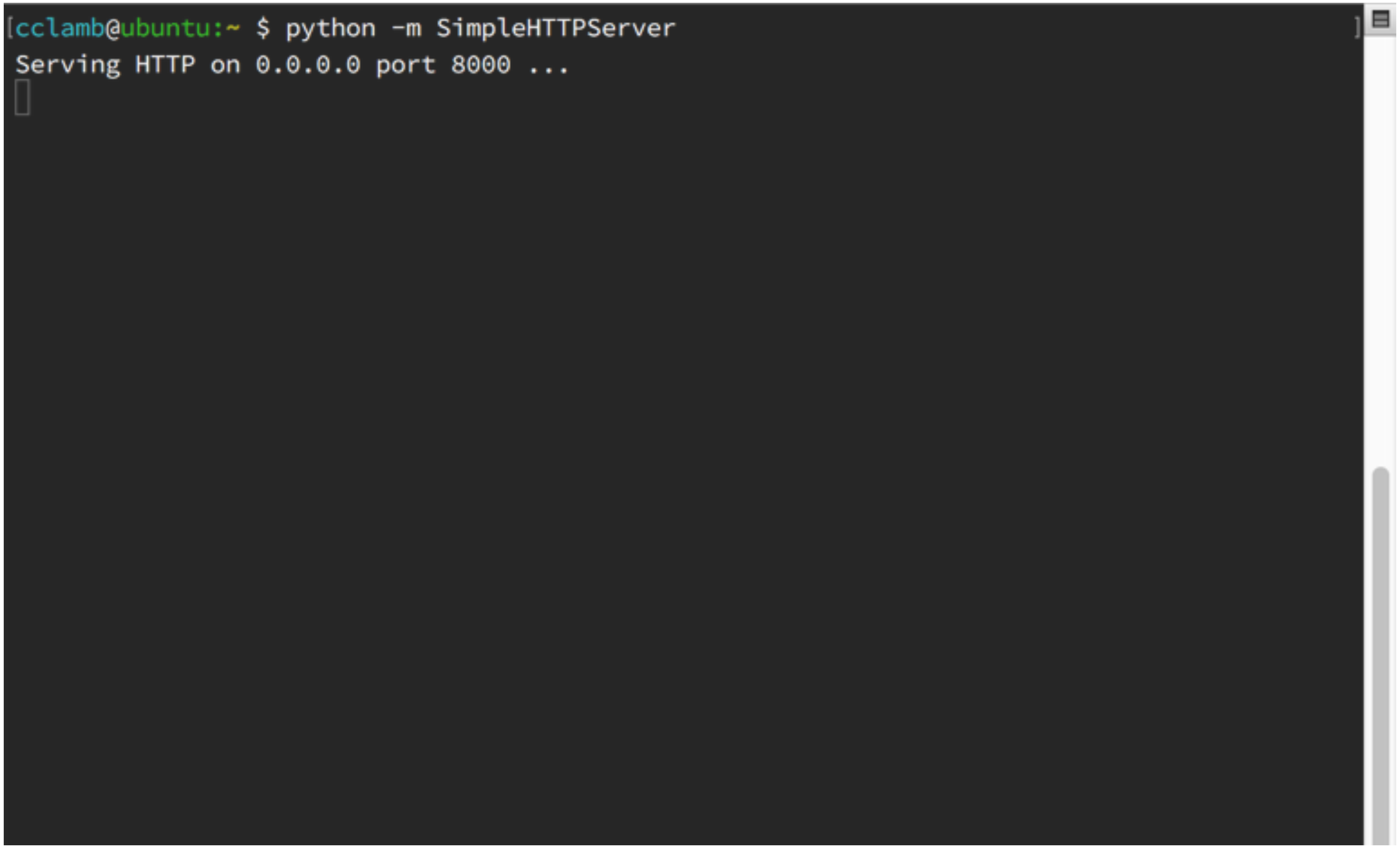
Why?

- A simple example
- Sends an HTTP GET

./requestor.c

```c
#include <stdio.h>
#include <curl/curl.h>

#define OK         0
#define INIT_ERR   1
#define REQ_ERR    2

#define URL        "http://localhost:8000"

int main(void) {
  CURL      *curl;
  CURLcode  res;

  curl = curl_easy_init();
  if (curl) {
    curl_easy_setopt(curl, CURLOPT_URL, URL);
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
    res = curl_easy_perform(curl);
    if(res != CURLE_OK) {
      return REQ_ERR;
    }
    curl_easy_cleanup(curl);
  } else {
    return INIT_ERR;
  }
  return OK;
}
```

NORMAL master ./requestor.c M unix



SimpleHTTPServer

$ python -m SimpleHTTPServer
(run this in another window)

Work — ssh -X 192.168.120.141 — 89×31

```
cclamb@ubuntu:~/Work/iot-client $ make
gcc   -c requestor.c -o requestor.o
gcc   -o test requestor.o -L/usr/lib/x86_64-linux-gnu -lcurl -lpthread
cclamb@ubuntu:~/Work/iot-client $ ls
makefile  printer.c  requestor.c  requestor.o  test
cclamb@ubuntu:~/Work/iot-client $ ./test > test.out
cclamb@ubuntu:~/Work/iot-client $
```

```
cclamb@ubuntu:~/Work/iot-client $ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [12/Jan/2017 16:55:17] "GET / HTTP/1.1" 200 -
```

```
2  0* bash   1h 51m 58s  0.0 0.0 0.1  2017-01-12  16:55  ubuntu
```

IoT: Client Devices

Moving onto ARM

```
CC=gcc
CCFLAGS=
INCLUDES=
LFLAGS=-L/usr/lib/x86_64-linux-gnu
LIBS=-lcurl -lpthread

SRC=requestor.c
OBJ=$(SRC:.c=.o)
MAIN=test

RM=rm -rf

.c.o:
	$(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

$(MAIN): $(OBJ)
	$(CC) $(CCFLAGS) $(INCLUDES) -o $(MAIN) $(OBJ) $(LFLAGS) $(LIBS)

all: $(MAIN)

clean:
	$(RM) $(MAIN) *.o *~
```

x86 Makefile

Usual Toolchain

```
BUILDROOT_HOME=/home/cclamb/buildroot-2016.11

CC=$(BUILDROOT_HOME)/output/host/usr/bin/arm-linux-gcc
CFLAGS=--sysroot=$(BUILDROOT_HOME)/output/staging
INCLUDES=
LFLAGS=

LIBS=-lcurl -uClibc -lc

SRC=requestor.c
OBJ=$(SRC:.c=.o)
MAIN=test

RM=rm -rf

.c.o:
	$(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

$(MAIN): $(OBJ)
	$(CC) $(CFLAGS) $(INCLUDES) -o $(MAIN) $(OBJ) $(LFLAGS) $(LIBS)

all: $(MAIN)
```

ARM Makefile

Surprise! Buildroot creates a toolchain for you!

Rebuild the Project

Make sure you have the right IP first

- requestor.c needs to have the IP address of your host

Check the BUILDROOT_HOME

- Needs to point to your BUILDROOT home directory

make -f makefile-arm

- …or use an alias: alias amake="make -f makefile-arm"

SCP and Test

SCP THE NEW PROGRAM TO YOUR ARM GUEST

- scp -P 2222 test user@localhost:~/

SIMPLEHTTPSERVER ON THE HOST

- python -m SimpleHTTPServer

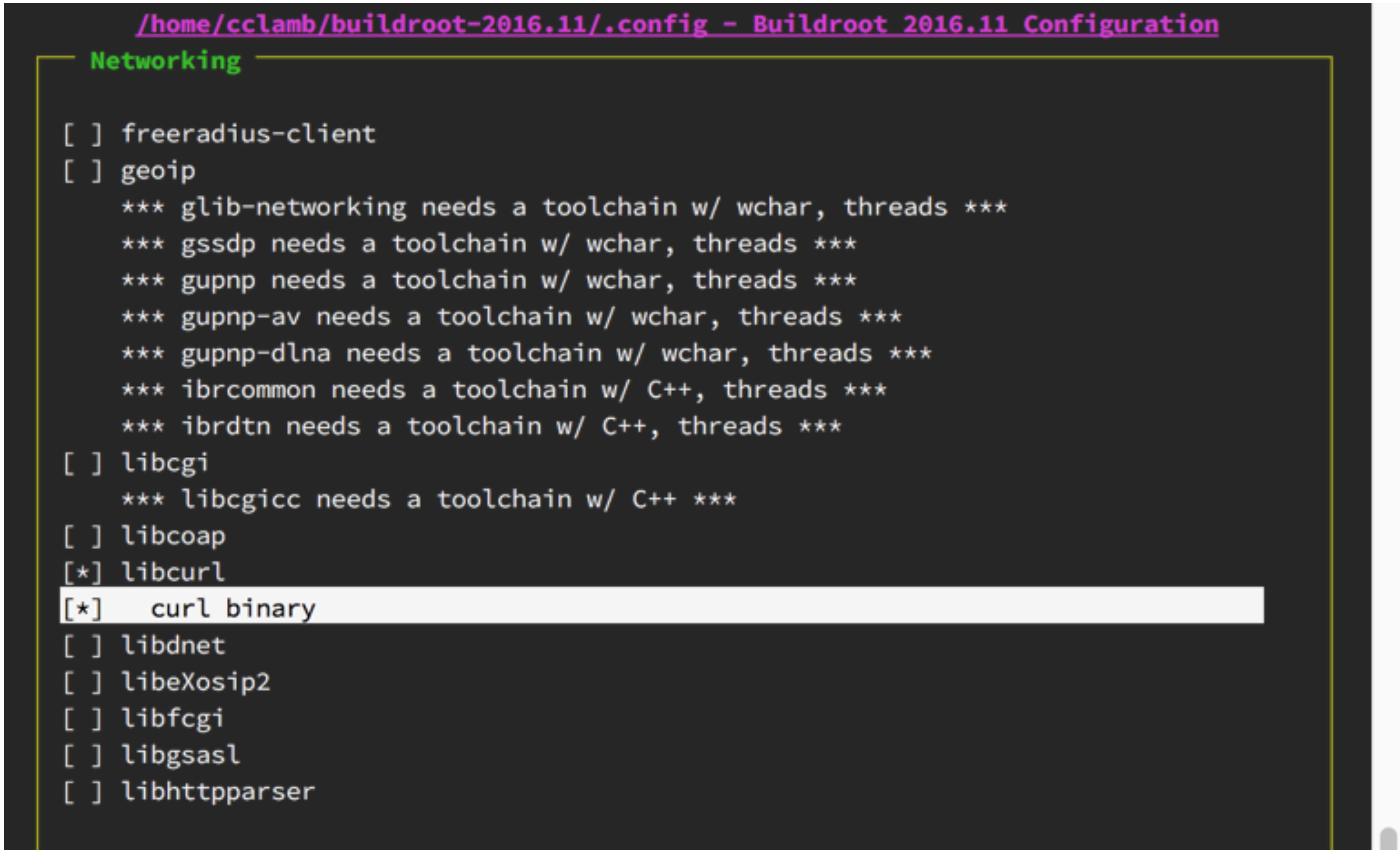
TEST FROM THE GUEST

- curl -v www.cnn.com,
- curl -v <host_ip_address>
- ./test

IoT: Client Devices

Networking Support and Buildroot

Remember to save your .config files!



libcurl

Target packages -> Libraries -> Networking

libcurl and curl

Why libcurl?

- Networking library
- Well supported and documented

Why curl?

- *Now we're making development changes!*
- We'll want to remove curl prior to deployment
- Very useful for network testing though

Build and Test Image

POST-BUILD CHANGES

- Add user account via **adduser**
  - **$ adduser -h /<home_dir> -s /bin/sh <username>**
- Edit /etc/shadow for new account
  - <user>:1Cw/pSYOL$16H24pCAifdyol19oRE1n1:**10933**:0:99999:7:::
- Test login
  - **$ ssh -p 2222 <user>@localhost**

```
$ which curl
/usr/bin/curl
$ ls /usr/lib
libcrypto.so        libcurl.so.4        libssl.so.1.0.0     libz.so.1.2.8
libcrypto.so.1.0.0  libcurl.so.4.4.0    libz.so
libcurl.so          libssl.so           libz.so.1
$
```

Take a Look Around

We have both /bin/curl and /usr/lib/libcurl.so

IoT: Client Devices

Networking Configurations

Networking in QEMU

A NUMBER OF NETWORKING MODES

- We’ll use SLIRP most (all?) of the time
- You can fall back to TAP mode, though we won’t discuss much

SLIRP IS SLOW

- Lots of overhead, not efficient
- Host has connectivity

TAP IS MUCH FASTER

- Host network adapter is bridged to Guest VM

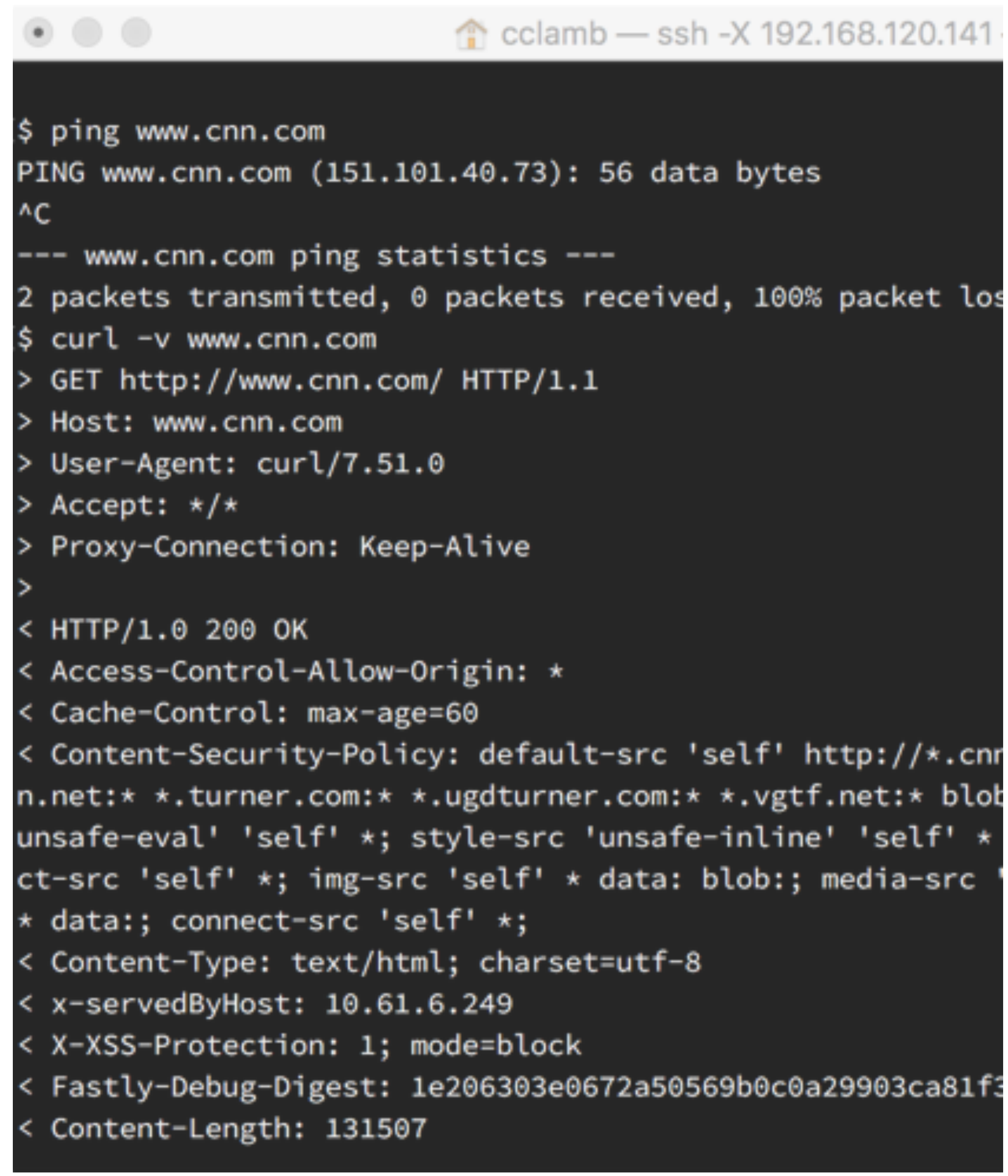
Bring up your QEMU

THIS CONFIG USES SLIRP

- Note, in the command that brings it up: **-net user**

BUT IT SEEMS TO WORK

- Ping doesn't work (you can configure ping to work though)
- Web access does work (we can test this thanks to **curl**)




```
$ ping www.cnn.com
PING www.cnn.com (151.101.40.73): 56 data bytes
^C
--- www.cnn.com ping statistics ---
2 packets transmitted, 0 packets received, 100% packet los
$ curl -v www.cnn.com
> GET http://www.cnn.com/ HTTP/1.1
> Host: www.cnn.com
> User-Agent: curl/7.51.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
< HTTP/1.0 200 OK
< Access-Control-Allow-Origin: *
< Cache-Control: max-age=60
< Content-Security-Policy: default-src 'self' http://*.cnn
n.net:* *.turner.com:* *.ugdturner.com:* *.vgtf.net:* blob
unsafe-eval' 'self' *; style-src 'unsafe-inline' 'self' *
ct-src 'self' *; img-src 'self' * data: blob:; media-src
* data:; connect-src 'self' *;
< Content-Type: text/html; charset=utf-8
< x-servedByHost: 10.61.6.249
< X-XSS-Protection: 1; mode=block
< Fastly-Debug-Digest: 1e206303e0672a50569b0c0a29903ca81f3
< Content-Length: 131507
```

```
[cclamb@ubuntu:~ $ ifconfig ens33                                                    ]
ens33     Link encap:Ethernet  HWaddr 00:0c:29:7f:9d:45
          inet addr:192.168.120.141  Bcast:192.168.120.255  Mask:255.255.255.0
          inet6 addr: fe80::e029:c10c:4763:4168/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1768 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1122 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:588198 (588.1 KB)  TX bytes:340479 (340.4 KB)

[cclamb@ubuntu:~ $ sudo python -m SimpleHTTPServer 80                                ]
Serving HTTP on 0.0.0.0 port 80 ...
```

HTTP to Host OS

Get IP on host and start Python SimpleHTTPServer

```
[$ curl -v 192.168.120.141                                                    ]
> GET / HTTP/1.1
> Host: 192.168.120.141
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/2.7.12
< Date: Wed, 11 Jan 2017 23:29:41 GMT
< Content-type: text/html; charset=UTF-8
< Content-Length: 2040
<
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a>
<li><a href=".bash_logout">.bash_logout</a>
<li><a href=".bashrc">.bashrc</a>
<li><a href=".binwalk/">.binwalk/</a>
<li><a href=".cache/">.cache/</a>
```

Use Curl on Guest

Send an HTTP GET request to the Host

```
[cclamb@ubuntu:~ $ ifconfig ens33
ens33     Link encap:Ethernet  HWaddr 00:0c:29:7f:9d:45
          inet addr:192.168.120.141  Bcast:192.168.120.255  Mask:255.255.255.0
          inet6 addr: fe80::e029:c10c:4763:4168/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1768 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1122 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:588198 (588.1 KB)  TX bytes:340479 (340.4 KB)

[cclamb@ubuntu:~ $ sudo python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
192.168.120.141 - - [11/Jan/2017 16:29:41] "GET / HTTP/1.1" 200 -
```

Check the Server

We received a request!

Why not TAP?

TAP WORKS FINE

- I’ve used it, but it requires more work on your part
- Configure local network topology
- Hijacks host adapter as a bridge endpoint
- Guest has IP from Host virtualization solution (in my case, VMWARE)
- Makes some of what we’re doing more difficult

```
#!/bin/sh

ip link add br0 type bridge
ip addr flush dev ens33

ip link set ens33 master br0

tunctl -u $(whoami)

ip link set tap0 master br0

ip link set dev br0 up
ip link set dev tap0 up

qemu-ifup (END)
```

```
#!/bin/sh

ip link set dev ens33 down
ip link set dev br0 down
ip link set dev tap0 down
ip link del dev br0
ip link del dev tap0
ip link set dev ens33 up
qemu-ifdown (END)
```

TAP Config Files

Run **qemu-ifup** to activate TAP on host, **qemu-ifdown** to tear down, change -**net user** to **-net tap,ifname=tap0**

IoT: Client Devices

Static v. Dynamic Details

Libraries

Static libraries

- .a suffix, contains object (.o) files, archived with **ar**
- Linked into program at *compile time*

Dynamic libraries

- .so (for *shared object*) suffix
- Linked into program at *execution* by the *loader*

Static Linking

Leads to large executables

- All the code is in the executable file

No code reuse

- All used functions must be in same file
- printf(.) code is in all files that print

Lots of wasted space

- We only need to store this code in a single place

```
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-nm test-print-s > nm_s.out
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-nm test-print-d > nm_d.out
cclamb@ubuntu:~/Work/iot-client $ ls -alh nm_*.out
-rw-rw-r-- 1 cclamb cclamb 971 Jan 26 17:27 nm_d.out
-rw-rw-r-- 1 cclamb cclamb 19K Jan 26 17:27 nm_s.out
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-nm test-print-d | grep puts
         U puts
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-nm test-print-s | grep puts
00010688 T fputs_unlocked
00010688 T __GI_fputs_unlocked
00010254 T puts
cclamb@ubuntu:~/Work/iot-client $
```

Comparing

Note the **U** and **T**

```
cclamb@ubuntu:~/Work/iot-client $ head nm_s_t.out
00010858 T abort
0001f8f8 T __adddf3
000201c8 T __aeabi_cdcmpeq
000201c8 T __aeabi_cdcmple
000201ac T __aeabi_cdrcmple
00020240 T __aeabi_d2uiz
0001f8f8 T __aeabi_dadd
000201dc T __aeabi_dcmpeq
00020218 T __aeabi_dcmpge
0002022c T __aeabi_dcmpgt
cclamb@ubuntu:~/Work/iot-client $ grep fopen nm_s_t.out
00019adc T fopen
00019adc T __GI_fopen
00019d84 T _stdio_fopen
cclamb@ubuntu:~/Work/iot-client $
```

Statically Linked

Statically linked executable contains *lots* of code, very little of which is used

IoT: Client Devices

Executable and Linking Format (ELF), a Quick Intro

What is it?

Various OS use various program formats

- Mach-o, PE, ELF are the big ones today

Ways to package programs

- Data, code, libraries, exported functions, etc.

Used for different things

- Object (.o) files, libraries (.so and .a files), executables, core dumps

Organization

Program & platform
readelf -h will show the contents

ELF Header

Program Header Table

Used to create a process; required in executables; show with *readelf -l*

Code lives here!

.text

.rodata

Read only data contained in this section

. . .

Initialized program data

.data

Section Header Table

Points to all the sections in the program image

```
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-readelf -r test-print-s

There are no relocations in this file.
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-readelf -r test-print-d

Relocation section '.rel.plt' at offset 0x2ac contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0002058c  00000216 R_ARM_JUMP_SLOT   00000000   puts
00020590  00000416 R_ARM_JUMP_SLOT   00000000   abort
00020594  00000516 R_ARM_JUMP_SLOT   00000000   __deregister_frame_inf
00020598  00000716 R_ARM_JUMP_SLOT   00000000   __uClibc_main
0002059c  00000c16 R_ARM_JUMP_SLOT   00000000   __register_frame_info
cclamb@ubuntu:~/Work/iot-client $
```

Procedure Lookup Table

Static has no relocations, dynamic does

```
000102f8 <puts@plt>:
   102f8:       e28fc600        add     ip, pc, #0, 12
   102fc:       e28cca10        add     ip, ip, #16, 20 ; 0x10000
   10300:       e5bcf28c        ldr     pc, [ip, #652]! ; 0x28c

00010304 <abort@plt>:
   10304:       e28fc600        add     ip, pc, #0, 12
   10308:       e28cca10        add     ip, ip, #16, 20 ; 0x10000
:

00010478 <main>:
   10478:       e92d4800        push    {fp, lr}
   1047c:       e28db004        add     fp, sp, #4
   10480:       e59f000c        ldr     r0, [pc, #12]   ; 10494 <main+0x1c>
   10484:       ebffff9b        bl      102f8 <puts@plt>
   10488:       e3a03000        mov     r3, #0
   1048c:       e1a00003        mov     r0, r3
   10490:       e8bd8800        pop     {fp, pc}
   10494:       000104a8        .word   0x000104a8
```

How do we relocate?

objdump -S test-print-d

IoT: Client Devices

Linking & Loading

Linkers

Creates Executable Images

- Libraries, executables, etc.

Uses object files

- .o files; you can see these when you build (usually)
- By extension, static libraries too (.a files)

Loaders

BOOTLOADERS, EMBEDDED SYSTEMS

- Bootloaders are special loaders, load OS/Kernel
- Embedded systems frequently do not have loaders
- We're using embedded linux though, which has one

LOADS PROGRAMS AND DYNAMIC LIBRARIES

- Loads programs into memory, starts execution (at _start)
- Sometimes uses a dynamic linker
- Executables use them

```
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-objdump -s test-print-d | head

test-print-d:     file format elf32-littlearm

Contents of section .interp:
 100f4 2f6c6962 2f6c642d 75436c69 62632e73  /lib/ld-uClibc.s
 10104 6f2e3000                             o.0.
Contents of section .hash:
 10108 03000000 0d000000 0a000000 0c000000  ................
 10118 09000000 00000000 00000000 00000000  ................
 10128 00000000 01000000 02000000 05000000  ................
cclamb@ubuntu:~/Work/iot-client $
```

ARM Dynamic Linker

Dynamic linker path is embedded in executable

Object Files

Object files contain object code

- Relocatable instructions for a platform
- Not directly executable

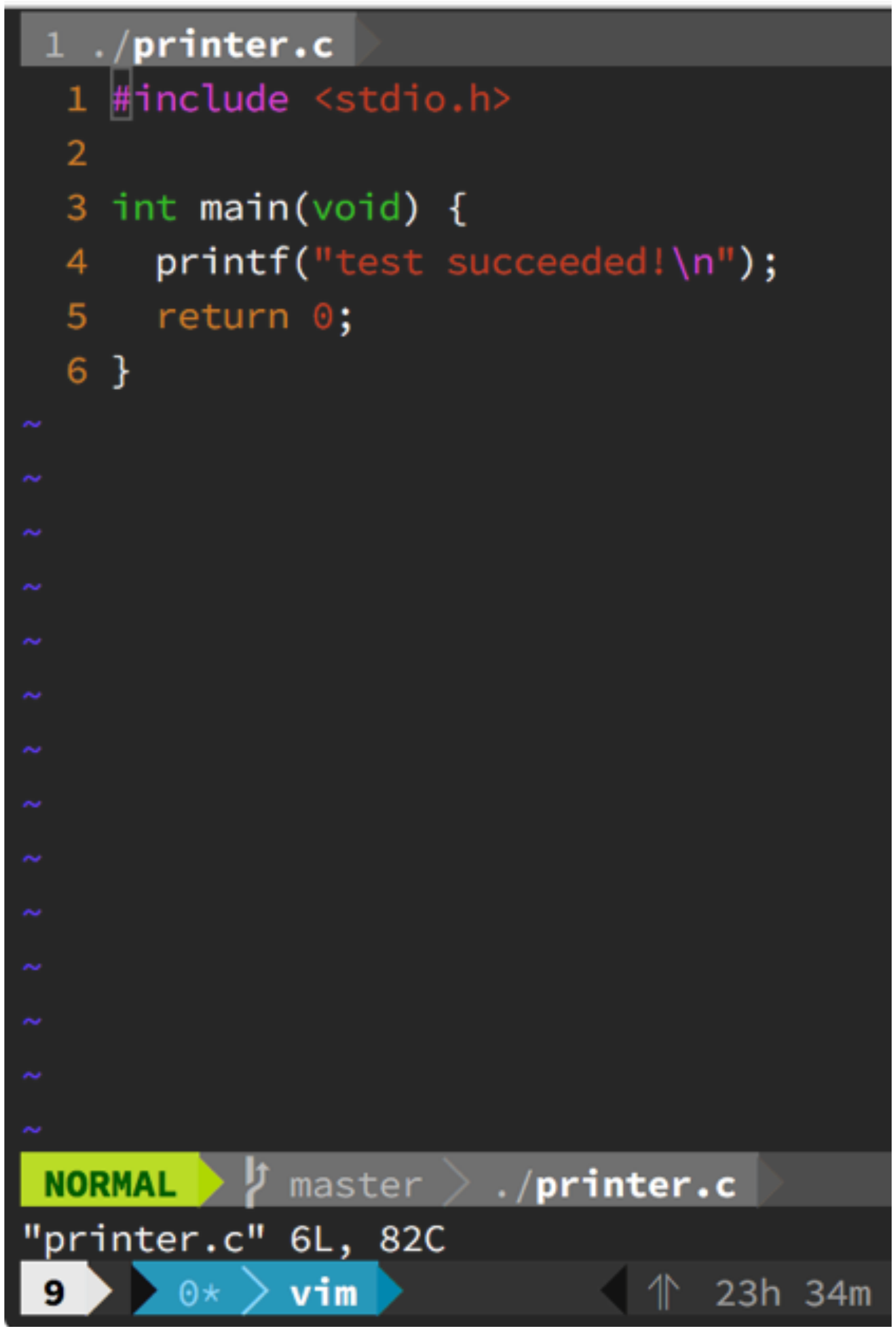
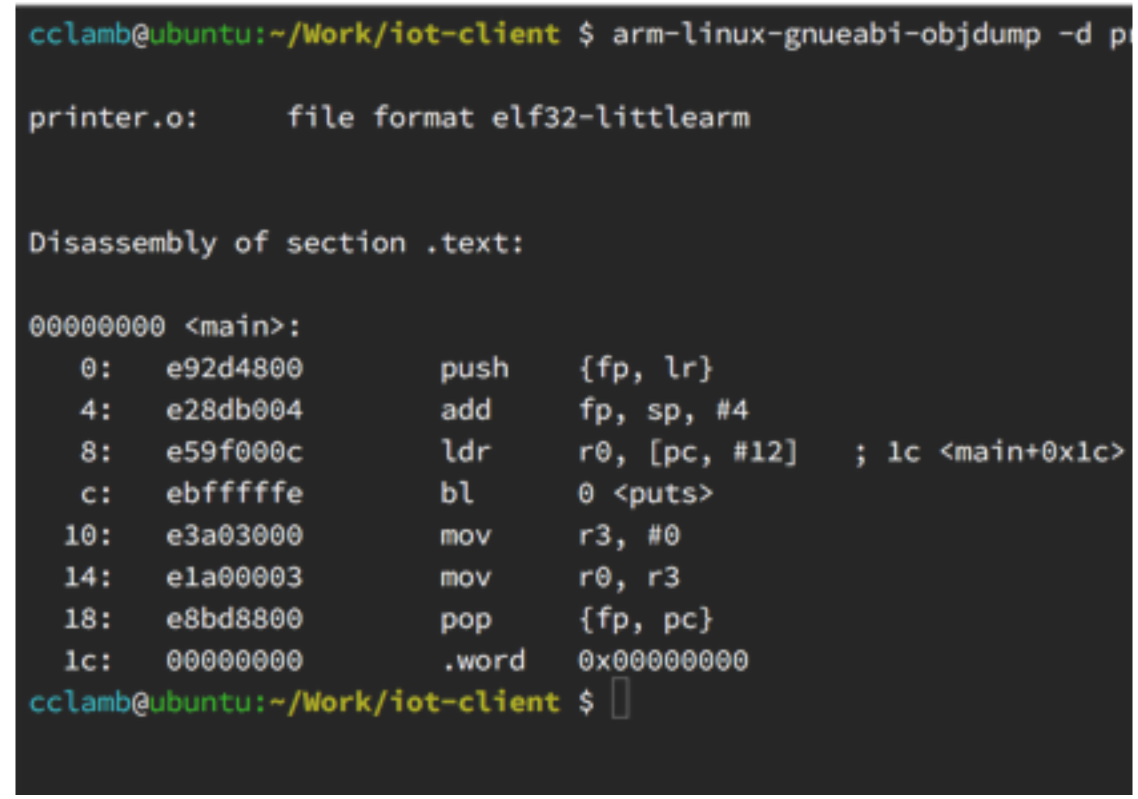
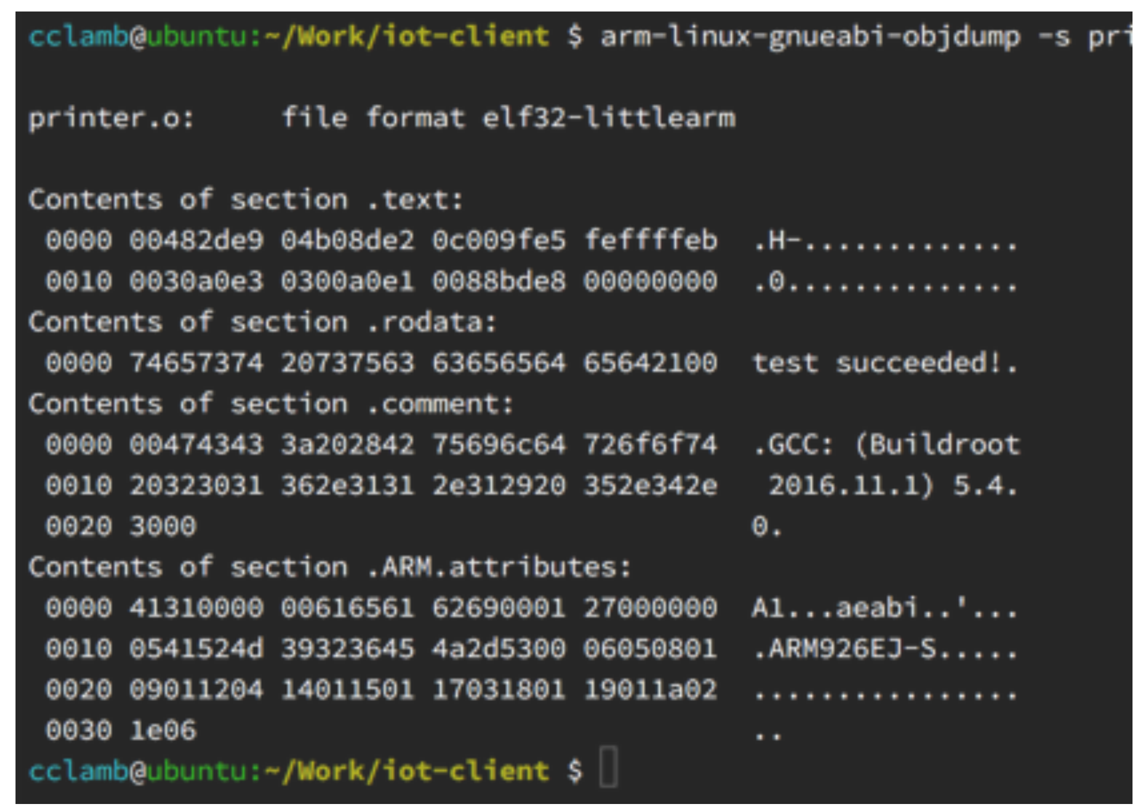
Relocatability is important

- The object code is inserted by the linker into a dynamic library or executable image
- Relocatability allows linker to place code arbitrarily (-ish)

```
cclamb@ubuntu:~/Work/iot-client $ sdmake
/home/cclamb/buildroot-2016.11.1/output/host/usr/bin/arm-linux-gcc --sysroot=/home/cclamb/b
uildroot-2016.11.1/output/staging  -c printer.c -o printer.o
/home/cclamb/buildroot-2016.11.1/output/host/usr/bin/arm-linux-gcc --sysroot=/home/cclamb/b
uildroot-2016.11.1/output/staging -static  -o test-print-s printer.o  -uClibc -lc
/home/cclamb/buildroot-2016.11.1/output/host/usr/bin/arm-linux-gcc --sysroot=/home/cclamb/b
uildroot-2016.11.1/output/staging  -o test-print-d printer.o  -uClibc -lc
cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-readelf -a printer.o > re.out
cclamb@ubuntu:~/Work/iot-client $ head re.out
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:                              REL (Relocatable file)
  Machine:                           ARM
  Version:                           0x1
cclamb@ubuntu:~/Work/iot-client $
```

Object File Example

Using our old printer example


1 ./printer.c
1 #include <stdio.h>
2
3 int main(void) {
4 printf("test succeeded!\n");
5 return 0;
6 }
NORMAL
master
./printer.c
"printer.c" 6L, 82C
9
0*
vim
23h 34m

cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-objdump -d pr
printer.o: file format elf32-littlearm
Disassembly of section .text:
00000000 <main>:
0: e92d4800 push {fp, lr}
4: e28db004 add fp, sp, #4
8: e59f000c ldr r0, [pc, #12] ; 1c <main+0x1c>
c: ebfffffe bl 0 <puts>
10: e3a03000 mov r3, #0
14: e1a00003 mov r0, r3
18: e8bd8800 pop {fp, pc}
1c: 00000000 .word 0x00000000
cclamb@ubuntu:~/Work/iot-client $

cclamb@ubuntu:~/Work/iot-client $ arm-linux-gnueabi-objdump -s pri
printer.o: file format elf32-littlearm
Contents of section .text:
0000 00482de9 04b08de2 0c009fe5 feffffeb .H-.............
0010 0030a0e3 0300a0e1 0088bde8 00000000 .0..............
Contents of section .rodata:
0000 74657374 20737563 63656564 65642100 test succeeded!.
Contents of section .comment:
0000 00474343 3a202842 75696c64 726f6f74 .GCC: (Buildroot
0010 20323031 362e3131 2e312920 352e342e 2016.11.1) 5.4.
0020 3000 0.
Contents of section .ARM.attributes:
0000 41310000 00616561 62690001 27000000 A1...aeabi..'...
0010 0541524d 39323645 4a2d5300 06050801 .ARM926EJ-S.....
0020 09011204 14011501 17031801 19011a02 ....
0030 1e06 ..
cclamb@ubuntu:~/Work/iot-client $

IoT: Client Devices

Static v. Dynamic Linking

1 ./printer.c

Bufs

```c
#include <stdio.h>

int main(void) {
  printf("test succeeded!\n");
  return 0;
}
```

Static Compilation

Same printer case from earlier

./makefile-arm-sd

```makefile
BUILDROOT_HOME=/home/cclamb/buildroot-2016.11.1

CC=$(BUILDROOT_HOME)/output/host/usr/bin/arm-linux-gcc
CFLAGS=--sysroot=$(BUILDROOT_HOME)/output/staging
INCLUDES=
LFLAGS=

LIBS=-uClibc -lc

SRC=printer.c
OBJ=$(SRC:.c=.o)
STATIC_MAIN=test-print-s
DYNAMIC_MAIN=test-print-d

RM=rm -rf

.c.o:
	$(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

all: $(OBJ)
	$(CC) $(CFLAGS) -static $(INCLUDES) -o $(STATIC_MAIN) $(OBJ) $(LFLAGS) $(LIBS)
	$(CC) $(CFLAGS) $(INCLUDES) -o $(DYNAMIC_MAIN) $(OBJ) $(LFLAGS) $(LIBS)

clean:
	$(RM) $(STATIC_MAIN) $(DYNAMIC_MAIN) *.o *~
```

Different Makefile

One dynamic (test-print-d), one static (test-print-s)

```
cclamb@ubuntu:~/Work/iot-client $ make -f makefile-arm-sd
/home/cclamb/buildroot-2016.11.1/output/host/usr/bin/arm-linux-gcc --sysroot=/home/cclamb
/buildroot-2016.11.1/output/staging -static  -o test-print-s printer.o  -uClibc -lc
/home/cclamb/buildroot-2016.11.1/output/host/usr/bin/arm-linux-gcc --sysroot=/home/cclamb
/buildroot-2016.11.1/output/staging  -o test-print-d printer.o  -uClibc -lc
cclamb@ubuntu:~/Work/iot-client $ ls -alh
total 188K
drwxrwxr-x  3 cclamb cclamb 4.0K Jan 16 18:09 .
drwxrwxr-x 18 cclamb cclamb 4.0K Jan 16 15:01 ..
drwxrwxr-x  8 cclamb cclamb 4.0K Jan 16 18:08 .git
-rw-rw-r--  1 cclamb cclamb   22 Jan 16 16:39 .gitignore
-rw-rw-r--  1 cclamb cclamb  427 Jan 16 16:26 makefile-arm
-rw-rw-r--  1 cclamb cclamb  552 Jan 16 17:57 makefile-arm-sd
-rw-rw-r--  1 cclamb cclamb  314 Jan 16 15:16 makefile-x86_64
-rw-rw-r--  1 cclamb cclamb   82 Jan 16 15:01 printer.c
-rw-rw-r--  1 cclamb cclamb 1.1K Jan 16 16:31 printer.o
-rw-rw-r--  1 cclamb cclamb  519 Jan 16 15:08 requestor.c
-rwxrwxr-x  1 cclamb cclamb 4.7K Jan 16 18:09 test-print-d
-rwxrwxr-x  1 cclamb cclamb 137K Jan 16 18:09 test-print-s
cclamb@ubuntu:~/Work/iot-client $
```

Run the Makefile

What are the differences?

Why so big?