ECE 471 – Embedded Systems
Lecture 26

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

3 November 2023

Announcements

- Project topics were due, I will respond to them
- Midterm #2 on Friday November 17th
- HW#9 will be posted, you can have two weeks as its bit harder, that does mean it's due same day as midterm

HW#9 – Summary

- Use a temperature probe (either SPI or 1-wire) and output the result to the i2c display
    - Re-use i2c display code from earlier homework
    - Re-use temp code (either TMP36 or the 1-wire)
    - Display the temperature on display
- When done can turn back in parts (assuming you aren't using them for the project)

HW#9 Notes – Modular Code

- In previous homeworks we put everything in one C file
- This isn't really practical for large projects
- By splitting things up into smaller files you can have some benefits:
  - Easier to organize/find code
  - Can re-use code easier
  - Less chance of merge conflicts when multiple people working on project in git
  - Can take common code and make libraries

HW#9 – Writing Modular Code

- In C you can compile each C file into its own object file, link together at end
- API defined in a header .h file
- For example in the homework, we could put temperature read code into its own file with a `double get_temperature(void)` interface
- For other C files to see this, you need to export the definition. Usually this is done by putting the advance definition `double get_temperature(void);` in a .h

header file and then including it in the other files

- Note: don't put full C functions in header files. I know this is a C++ thing but it's usually frowned upon when programming in C
- Each file does not need a `main()` function, you only need one per combined program.

HW#9 – Building Modular Code

- To link the various .o files together involves the "linker". However it's easier to just let gcc do it (gcc knows how to run the linker for you) `gcc -o display_temp display.o temperature.o`
- The linker merges the .o files into one big executable, and makes sure the placeholders to functions/variables in all of the files get the right addresses/pointers to where things live in the finished executable.
- How do you make sure when you change one C file that

everything that uses it is also rebuilt? A well-crafted Makefile will have all these dependencies in place and will rebuild everything properly.

HW#9 – Converting Floating Point to Digits

- Use sprintf()

```
char string[128];
double temperature;
sprintf(string,"%.1lf",temperatur
/* Now string[0] has first digit,
```

- Use division/modulus

```
double temperature=23.4;
int hundreds,tens, ones,remainder
```

```
hundreds=temperature/100;
remainder=temperature%100;
tens=remainder/10;
ones=remainder%10;
```

HW#9 – Writing Good Testcases

- Once you have written your nice modular code, how can you test it?
- Need to write some test cases that test a wide range of behaviors
- In the homework I have you think up some test cases

Types of Security Compromise

- Crash
  “ping of death”
- DoS (Denial of Service)
- User account compromise
- Root account compromise
- Privilege Escalation
- Rootkit
- Re-write firmware? VM? Above OS?

Information Leakage / Side Channel Attacks

- Can leak info through side-channels
- Detect encryption key by how long other processes take? Power supply fluctuations? RF noise?
- Timing attacks
- If code takes different paths through code can notice this via linked info

  Solution: cycle-invariant code, takes same amount of time for all paths through code (really hard to write code like this)

- Recent CPU architecture extensions to help with this (ARM64 DIT data independent timing)

Information Leakage: Meltdown and Spectre

- Can use timing to find if address is in cache
- If speculative execution, can do things like

```
if (secret&1) a[0]=1;
else a[4096]=1;
```

then use timing to see which one was brought in

Deceptive Code

- Can you sneak purposefully buggy/exploitable code into open source?
- Can you sneak bad code (or use typo-squatting) to trick people in large public repositories (like javascript/npm)
- To-do at U of Minnesota where researches tried (unsuccessfully it turns out) to sneak questionable code into the kernel
- "Trojan Source" in the news: can use unicode (including

left-right reversal) to have code that looks correct but compiler will compile differently x!=y vs y=!x

- Should code allow non-ASCII?

to apply updates

Finding Bugs

- Source code inspection
- Watching mailing lists
- Static checkers (coverity, sparse)
- Dynamic checkers (Valgrind). Can be slow.
- Fuzzing

perf_fuzzer

- Fuzzers intentionally try invalid/dangerous input by generating random inputs causing crash
- I wrote the `perf_fuzzer` which found many bugs in Linux kernel with the perf_event_open() syscall

Reporting Bugs

- So you found a security bug...
- Who do you contact?
- What's responsible disclosure?
- Bug bounties
- Can be a hassle reporting properly, and companies are always suspicious and can even accuse you of evil hacking

Computer Security

Social Engineering

- Often easier than actual hacking
- Talking your way into a system
- Looking like you know what you are doing
- “The Art of Deception”