ECE 471 – Embedded Systems
Lecture 26

Vince Weaver

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

vincent.weaver@maine.edu

8 November 2024

Announcements

- HW#8 is due
- Project topics are due
- Midterm #2 on Wednesday November 20th
- HW#9 will be posted, you can have two weeks as it's a bit harder
- No Class Monday

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.

- What if you want to make an official library? Static libraries are .a, dynamic .so. It's fairly easy to do this, just a few extra command line tools like `ar` or maybe even just using `-shared` to gcc

HW#9 – Converting Floating Point to Digits

- Use sprintf()

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

- Use division/modulus

```c
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

Computer Security
and why it matters for embedded systems

- Most effective security is being unconnected from the world and locked away in a box. Until recently most embedded systems matched that.

- Modern embedded systems are increasingly connected to networks, etc. Embedded code is not necessarily prepared for this.

- Internet of Things: IoT (the S is for Security)

Computer Security – The Problem

- Untrusted inputs from user can be hostile.
- Users with physical access can bypass most software security.

What can an attacker gain?

- Fun / Mischief
- Profit
- A network of servers that can be used for illicit purposes (SPAM, Warez, DDOS, bitcoin mining)
- Spying on others (companies, governments, etc)

Sources of Attack

- Untrusted user input
  Web page forms
  Keyboard Input

- USB Keys (CD-ROMs)
  Autorun/Autostart on Windows
  Scatter usb keys around parking lot, helpful people plug into machine.

- Network

cellphone modems
ethernet/internet
wireless/bluetooth

- Backdoors
  Debugging or Malicious, left in place

- Brute Force – trying all possible usernames/passwords

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?

Unsanitized Inputs

- Using values from users directly can be a problem if passed directly to another process
- If data (say from a web-form) directly passed to a UNIX shell script, then by including characters like ; can issue arbitrary commands: `system("rm %s\n",userdata);`
- SQL injection attacks; escape characters can turn a command into two, letting user execute arbitrary SQL commands; xkcd `Robert '); DROP TABLE Students;--`

https://xkcd.com/327/

Buffer Overflows

- User (accidentally or on purpose) copies too much data into a fixed sized buffer.

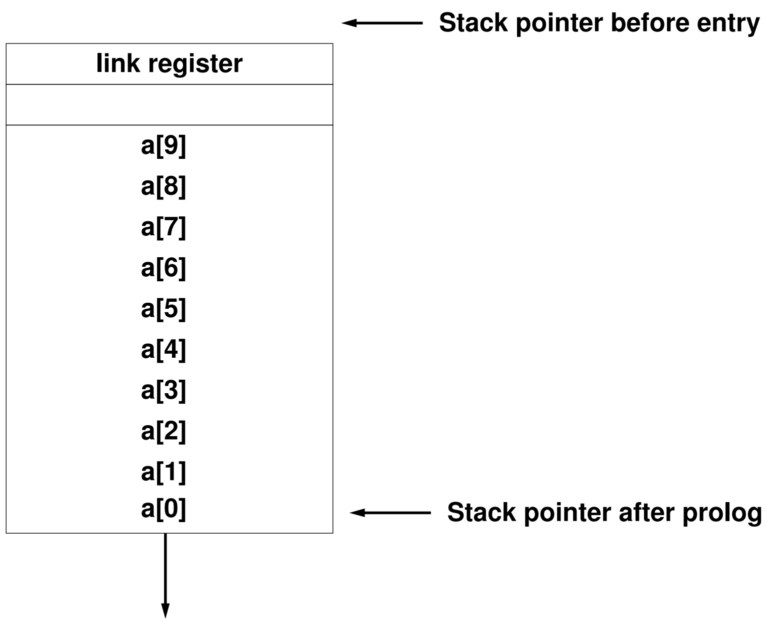
- Data outside expected area gets over-written. This can cause a crash (best case) or if user carefully constructs code, can lead to user taking over program.

Buffer Overflow Example

```
void function(int *values, int size) {
    int a[10];

    memcpy(a,values,size);

    return;
}
```

Maps to

```
    push    {lr}
    sub sp,#44

    memcpy

    add sp,#44
    pop {pc}
```



A value written to a[11] overwrites the saved link register. If you can put a pointer to a function of your choice there you can hijack the code execution, as it will be jumped to at function exit.

Mitigating Buffer Overflows

- Extra Bounds Checking / High-level Language (not C)
- Address Space Layout Randomization
- Putting lots of 0s in code (if strcpy is causing the problem)
- Scanning for unusual characters (can you write all-ASCII shellcode?)
- Running in a "sandbox"

Coding Mistakes with Security Implications

Dangling Pointer / Null Pointer Dereference

- Typically a NULL pointer access generates a segfault
- If an un-initialized function pointer points there, and gets called, it will crash. But until recently Linux allowed users to `mmap()` code there, allowing exploits.
- Other dangling pointers (pointers to invalid addresses) can also cause problems. Both writes and executions can cause problems if the address pointed to can be mapped.

Privilege Escalation

- If you can get kernel or super-user (root) code to jump to your code, then you can raise privileges and have a "root exploit"
- If a kernel has a buffer-overrun or other type of error and branches to code you control, all bets are off. You can have what is called "shell code" generate a root shell.
- Some binaries are setuid. They run with root privilege but drop them. If you can make them run your code before dropping privilege you can also have a root exploit.

◦ ping (requires root to open raw socket)
◦ X11 (needs root to access graphics cards)
◦ web-server (needs root to open port 80).

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?